Design Pattern Rationale Graphs: Linking Design to Source

by

Elisa L. A. Baniassad

M.Sc. University of British Columbia, 1997,
B.Sc. Technical University of Nova Scotia, 1995

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

December 2002

© Elisa Baniassad, 2002
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of **Computer Science**

The University of British Columbia
Vancouver, Canada

Date **December 3, 2002**
Abstract

As source code evolves, the reasoning behind certain design decisions is lost, often leading to violations of design goals during program maintenance. This kind of code decay is often caused by changes made by developers who are not adequately aware of the rationale behind the source code. Becoming aware of rationale is difficult for many reasons, including the mismatch between design and source code.

The thesis of this research is that a mechanism for tracing design goals through existing documentation to source would enable software developers to have more complete knowledge of design goals relevant to a system, and more confidence in terms of how design goals relate to the source. The technique should be applicable within the context of one software evolution task. To validate the claims of this thesis, we have developed the Design Pattern Rationale Graph (DPRG) approach and associated tool. This dissertation describes this mechanism, and its use in the validation of the thesis claims.

A DPRG is a graph formed from the text of design pattern documentation that is linked to a code base. The DPRG allows developers to view design pattern information related to design goals separately, and trace those goals through to their corresponding locations in a code base. The DPRG approach is lightweight: The time and effort required to apply the approach fits within the context of a single task.

We demonstrate the validity of the thesis claims by applying the DPRG in several case studies and one experiment. These studies address each claim separately, and do so on a wide range of systems and tasks.
Contents

Abstract ii

Contents iii

List of Tables viii

List of Figures ix

Acknowledgements xi

1 Introduction 1

1.1 Why is Rationale Awareness Difficult? 3

1.2 Current Rationale Elucidation Techniques 5

1.2.1 Encode Design in Code 6

1.2.2 Compare Design to Code 7

1.2.3 Trace Design Concepts to Code 8

1.3 Overview of the DPRG Approach and Tool 9

1.3.1 DPRG Structure 9

1.3.2 An Introduction to the DPRG Tool 10

1.4 Organization of the Dissertation 17

2 Preliminary Studies 18

2.1 Study: Developers and Concerns 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.1 Design</td>
<td>21</td>
</tr>
<tr>
<td>2.1.2 Results</td>
<td>24</td>
</tr>
<tr>
<td>2.2 Study: Detail and Context Reporting</td>
<td>33</td>
</tr>
<tr>
<td>2.2.1 Design</td>
<td>33</td>
</tr>
<tr>
<td>2.2.2 Results</td>
<td>36</td>
</tr>
<tr>
<td>2.3 Summary</td>
<td>41</td>
</tr>
<tr>
<td>3 Design Pattern Rationale Graph Model</td>
<td>42</td>
</tr>
<tr>
<td>3.1 Pattern-Level</td>
<td>42</td>
</tr>
<tr>
<td>3.1.1 Design Patterns as Pattern-Level Graphs</td>
<td>44</td>
</tr>
<tr>
<td>3.1.2 Sentence Chains</td>
<td>45</td>
</tr>
<tr>
<td>3.1.3 Sequence Chains</td>
<td>50</td>
</tr>
<tr>
<td>3.2 Source-Level</td>
<td>50</td>
</tr>
<tr>
<td>3.3 Link-Level</td>
<td>52</td>
</tr>
<tr>
<td>3.3.1 Code-Facts</td>
<td>55</td>
</tr>
<tr>
<td>3.3.2 Links to Source-Level Nodes</td>
<td>55</td>
</tr>
<tr>
<td>3.3.3 Source-Level Nodes and Edges</td>
<td>58</td>
</tr>
<tr>
<td>3.4 Summary</td>
<td>58</td>
</tr>
<tr>
<td>4 Design Pattern Rationale Graph Tool</td>
<td>60</td>
</tr>
<tr>
<td>4.1 Preparation of Input Files</td>
<td>60</td>
</tr>
<tr>
<td>4.1.1 Pattern-Level Files</td>
<td>60</td>
</tr>
<tr>
<td>4.1.2 Source-Level Files</td>
<td>61</td>
</tr>
<tr>
<td>4.1.3 Link-Level Files</td>
<td>62</td>
</tr>
<tr>
<td>4.2 Interaction with the DPRG Tool</td>
<td>62</td>
</tr>
<tr>
<td>4.2.1 Performing Queries on a DPRG</td>
<td>65</td>
</tr>
<tr>
<td>4.2.2 Linking the Pattern and Source Model Levels</td>
<td>70</td>
</tr>
<tr>
<td>4.3 DPRG Computation</td>
<td>74</td>
</tr>
<tr>
<td>4.3.1 Forming the Pattern-Level Graph</td>
<td>75</td>
</tr>
</tbody>
</table>
6.1.2 Support for Overlapping Patterns ........................................ 138
6.1.3 Pattern-Level Reusability ...................................................... 139
6.1.4 Applicability of the DPRG Model to Patterns in General .... 140
6.1.5 Attributing Empirical Results to Approach or Tool ........... 141

6.2 Use ......................................................................................... 142
6.2.1 Performing Queries Separately at the Pattern-Level .......... 142
6.2.2 Robustness against Varying User Capabilities ................. 143
6.2.3 Code-Fact Alteration to Enhance Linking Flexibility ... 143
6.2.4 Correspondence Rating Limitations ................................. 144

6.3 Implementation ....................................................................... 146
6.3.1 Lexical Indicators for Linking .............................................. 146
6.3.2 Potential Use of Pattern Finding Technologies ............... 146
6.3.3 Synonym Resolution .......................................................... 146
6.3.4 Pronoun Resolution .......................................................... 148

6.4 Summary .................................................................................. 149

7 Related Work ............................................................................ 151
7.1 Encode Design Concepts in Code ............................................ 151
7.2 Analysis Between Design and Code ....................................... 153
7.2.1 Verify Design ........................................................................ 153
7.2.2 Extrapolating Design .............................................................. 154
7.2.3 Generate Code from Design .................................................. 157
7.3 Follow Links from Design to Code .......................................... 159
7.3.1 Knowledge Bases ................................................................. 159
7.3.2 Design Rationale Tracing ...................................................... 160
7.4 Express Crosscutting Concerns in Design ......................... 163
7.5 Pattern-Level Related Work .................................................... 164
7.5.1 Concept Graphing ............................................................... 164
7.5.2 Concept Abstraction ............................................................ 165
List of Tables

2.1 Program Change Tasks and Strategies .................................. 26
2.2 Confidence and Correctness of Participant Responses ............. 38

3.1 Source-Level Edges ................................................................. 53

4.1 Definitions Used in Inference and Correspondence Rating ........ 80

5.1 Patterns Used in Validation ..................................................... 89
5.2 Design goals Identified for the CACHING Pattern .................. 96
5.3 Design goals Identified for the THREAD-SPECIFIC STORAGE Pattern 97
5.4 Design Goals Identified for the STRATEGY Pattern ................. 98
5.5 Actions of the Lightweightness Subject 1 .............................. 116
5.6 Actions of the Lightweightness Subject 2 .............................. 117
5.7 Summary of Lightweightness Results .................................... 119
5.8 Confidence and Correctness of Control Group Participant Responses 130
List of Figures

1.1 Three Levels of a DPRG ........................................... 10
1.2 An Abstract Depiction of the OBSERVER Pattern ............... 11
1.3 Three queries on the OBSERVER/JHotDraw DPRG ............... 13
1.4 Expansion of the pull model node ............................... 15
1.5 Expansion of notif at pattern-level and link-level ............... 16

2.1 Obstacle Types: Core/Concern Intersection ..................... 28
2.2 Code Alterations show Inward Reasoning ....................... 29
2.3 Code Containing Data Structure Assumptions .................. 31

3.1 Refined View of the DPRG Model ............................... 43
3.2 Abstract View of a Sentence Chain Structure .................. 46
3.3 How to Read a Sentence Chain .................................. 51
3.4 Abstract Representation of a Sequence Chain .................. 52
3.5 Abstract Representation of the Source-Level ................... 54
3.6 Portion of the OBSERVER Code-fact Graph ..................... 56
3.7 Links from Pattern-level Entities to Source-level Entities .... 57
3.8 Portion of the Link-level for the OBSERVER Pattern .......... 59

4.1 Class Diagram for the OBSERVER Pattern ...................... 63
4.2 Code-fact File for the OBSERVER Pattern ..................... 64
4.3 User Interaction Model for the DPRG Tool ...................... 65
4.4 User Interaction Model for Performing Queries on a DPRG 66
4.5 Regular Expression Query for efficien 67
4.6 Link-level Expansion of subject Node 69
4.7 User Interaction Model for Forming the Link-level of a DPRG 70
4.8 Initial Seed Input to the DPRG tool 71
4.9 Link Inspection 74
4.10 DPRG Representation of a Sentence 76
4.11 Algorithm to Compute a Node Expansion 78
4.12 Edges Between P and S Show Inferred Links 79
4.13 Sets Used in Calculating a Correspondence Rating for a Link (pe, se) 81

5.1 Investigation Path for Subject 1 104
5.2 Investigation Path for Subject 2 107
5.3 Detail Reported by REACTOR Subjects 127
5.4 Subjects’ Investigation Paths 133
Acknowledgements

This work is really thanks to the many people who helped along the way. First and foremost, I'd like to thank my supervisor, Gail Murphy, for her tireless support and work. There are so many times through this process I've been amazed by and grateful to Gail that they're really impossible to count. Somehow she guided me through what seemed like bleak times, and got me through what really were rushed times, and managed to do it all with such an air of ease and humor. Wow. Thanks.

I'd also like to thank Christa Schwaningger, my partner in crime at Siemens. Christa was responsible for finding the case studies used for the validation of this thesis, but also acted as friend and sounding board for my stay there. This could not have happened without you. Thanks also go to my supervisory committee: Norm Hutchinson, Cristina Conati, and Kris deVolder. Your comments greatly improved the quality of the thesis, as did the many words of support and encouragement.

Thanks to my husband Ryan, for his wonderful friendship, support and care through these many years, and especially for his work as editor and audience in the last few months. I'd also like to send my eternal gratitude to my parents, who were willing to stay up late nights re-checking drafts of my thesis, or even just chatting about it.

Thanks go to Yvonne Coady, for reading an early draft of the thesis, and to Jonathan Sillito, Joon Suan Ong and Yvonne for listening to endless versions of my defense talk, and for always asking tough questions but always with big smiles. I'd also like to thank all the people who participated in the case studies: the developers at Siemens, and the students at UBC.

Elisa L. A. Baniassad

The University of British Columbia
December 2002
Chapter 1

Introduction

*Law of continuing change.* A system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it …

*Law of increasing entropy.* The entropy of a system (its unstructuredness) increases with time, unless specific work is executed to maintain or reduce it …

*Belady and Lehman, The Laws of Program Evolution Dynamics [7]*

Software is the embodiment of a system’s design goals. Over time, goals are added, changed, and removed. Systems may need to be made faster, more user-friendly, or smaller. They may need to cater to new hardware, new software, or new domains of use. These are the kinds of changes that underly the process of software change, or evolution, as described in the First Law of Program Evolution Dynamics [7].

As goals are introduced for a system, the second law comes into effect: Structure decays, and maintenance becomes more difficult. In their study of code decay [20], Eick et al. noted several reasons why code becomes harder to change as its life progresses, including the introduction of changes that may violate the system’s original design principles.

The decay of a system under change may be reduced if developers understand
the design goals motivating the changed code: the rationale behind the code. If code is changed without accounting for all the effects of the change, the system’s design may be compromised. Parnas used the term ignorant surgery to refer to changes made to a software system by developers who do not understand the original design concepts behind the code [53]. When ignorant surgery occurs, the structure of a system tends to degrade, leading to design inconsistency, and increased costs for subsequent evolutionary tasks.

To assist developers in becoming informed of design goals motivating their source code, several approaches have been proposed.

First, techniques have been suggested that allow developers to look in the source code itself for design information (Section 1.2.1). These techniques involve encoding design information into the source code, and hence require either preparation work by early developers of a code base, or significant analysis by later developers.

Second, tools have been introduced which allow developers to analyze their code with relation to design (Section 1.2.2). Such tools use mechanisms that verify the structure of code or generate design abstractions from the source code. These mechanisms, however, do not provide the reasoning behind the structure.

Third, design rationale approaches are intended to help developers trace goals down to source (Section 1.2.3). In general, these approaches pose difficulty because recording the rationale for each code base necessitates a large effort on the part of a developer.

The thesis of this research is that a mechanism for tracing design goals through existing documentation to source would enable software developers to have more complete knowledge of design goals relevant to a system, and more confidence in terms of how design goals relate to the source. To validate the claims of this thesis, we have developed the Design Pattern Rationale Graph (DPRG) approach and associated tool. This dissertation describes this mechanism, and its use in the validation of the thesis claims.
A DPRG enables a developer to explore design documentation and relate portions of a code base to the design goals that motivate it. In doing so, it is intended to address the main problems outlined in the thesis: allowing a developer to become more informed of the rationale behind their code, leading to less structural degradation, and enabling more cost-effective changes. A DPRG is a graphical representation of the text of design documentation linked to a model of a code base. It encodes and depicts the relationships between design entities described in the design text. The DPRG allows the reader to perform their investigation separately from the intended structure of the document, and affords the reader an abstraction from the text. The DPRG tool is a lightweight mechanism, in that the amount of work required by a developer to apply the approach can be cost effectively applied in the context of a task.

The DPRG represents design documentation as found in design patterns [23]. A design pattern is a high-level abstraction for a commonly used solution to a commonly encountered software design problem. The solution a design pattern describes is not tied to one specific implementation: it may be found or applied in many systems. In addition to the design solution itself, a design pattern describes rationale for design decisions.

To validate the thesis claims, we present several studies. Two studies address the main claims: The DPRG approach affords developers more complete knowledge of, and more confidence with relation to, design goals. Other studies verify the lightweightness of the approach, and the efficacy of the DPRG representation.

1.1 Why is Rationale Awareness Difficult?

The entropy of a software system might be reduced if developers understood the rationale, or motivating design goals, behind the code. However, there are barriers to this awareness.

\[\text{1For an in-depth description of design patterns, please refer to Appendix A.}\]
In an ideal situation, the source code would correspond to design. In practice, however, many goals can not be represented by the source code that implements them. Some design goals are abstract, like "usability", or "maintainability"; others are more concrete, like "blocking" or "memory allocation". Typically, the more abstract a goal is, the larger its scope of effect on a system. Goals also map differently to source code: Concrete goals like "memory allocation" may be represented directly in certain programming languages, whereas a goal such as "usability" might not be directly expressible.

Design goals that cannot be directly represented in a programming language may become tangled in a code base. This may occur when a module of a software system simultaneously interacts with more than one goal. It may also happen that the code implementing a design goal becomes scattered throughout the code base. As has been pointed out by Soloway et al. [46], developers must deal with the results of delocalized plans: Design decisions whose consequences are spread over multiple parts of a program. Goals that cannot be directly represented in a programming language have been referred to as crosscutting concerns [71]. Crosscutting concerns make it difficult for a developer to associate design rationale with particular portions of a software system.

If code is not sufficient, perhaps the pain of software evolution could be eased through documentation [53, 7]. To better understand the reasoning behind the structure of a system, developers could examine the documentation to look for rationale context. Unfortunately, documentation of the design of a code base is often out of date because the documentation is not changed to reflect the code base. A developer examining stale documentation is likely to face difficulties associating it with the current structure of the system.

Some documentation escapes becoming stale by describing a general solution rather than a particular part of a system. A design pattern, for instance describes a general solution rather than a particular code base. For this reason, developers
examining systems that make use of design patterns can read the descriptions to learn more about the structure, behaviour, and goals of the portions of the code base that implement the design pattern solution. However, even when the documentation is up-to-date, or generally applicable, there may still be difficulty in becoming aware of the rationale which it describes.

Design patterns, for instance, are written with great care; however, aligning the solutions presented in design patterns with a particular implementation is not always straightforward. Within a design pattern solution, options are often presented. The Observer pattern description [23], for instance, is eleven pages long and presents numerous implementation choices. To determine how goals described in the design pattern are carried out in a code base it is necessary to determine which options were used, and which design goals are relevant for those options. Additionally, design patterns often address more than one goal. These goals may be spread throughout, or crosscut, the text of the pattern. For instance, in the Observer pattern, the concept of efficiency is mentioned in three separate places. When concepts, or themes are spread throughout a document, they become harder to understand [25]. Re-writing a design pattern to encapsulate each concept would, however, be unreasonable and would likely detract from the quality of the design description.

Mechanisms to assist a developer in maintaining rationale awareness must allow for the crosscutting nature of goals throughout a code base, must be flexible with relation to the levels of abstraction of the goals, must present information related to goals in a well-structured way, and must enable a developer to access rationale information within the time constraints of software maintenance tasks.

1.2 Current Rationale Elucidation Techniques

In general, there are three proposed ways for developers to increase their awareness of design rationale: by encoding design in code, by comparing and analyzing design and
code, and by following links from design to code. An overview of these categories is provided below and examples of approaches are given; more information is provided in Chapter 7.

1.2.1 Encode Design in Code

There are several ways in which design information can be encoded into source. Two such techniques are coding conventions, as in information transparency [26], and in-code documentation, such as that provided by literate programming [38].

In information transparency [26], programmers encode design information by through coding conventions. Tools such as Griswold's Aspect Browser [27] can then be used to search through the code base to expose portions pertaining to a particular concept. The application of information transparency is best done at the time of program creation, since means such as naming conventions are easiest to employ when integrated early. Because concerns are encoded through coding conventions, information transparency is limited in terms of how it can present the rationale behind design choices.

Literate programming [38] enables developers to integrate the description of design goals and rationale into source code. In addition to the original, WEB [38], many literate programming tools are available. One such tool is CWEB [39], which is a preprocessor to support literate programming in C and C++. In CWEB, programmers use mark-up language features to distinguish between different sections of their program. These sections can be separated and manipulated or compiled separately. As with information transparency, in literate programming conceptual design information is likely best applied either at the time of coding, or by someone intimately familiar with the code base.

These techniques require prescience of the design concepts of interest. Because of the effort associated with gathering the requisite knowledge, developers are unable to apply these techniques to capture all design concepts.
1.2.2 Compare Design to Code

In attempting to resolve design goals for a system with its implementation, developers can compare information gathered from analysis of the code with a design they believe carries out certain design goals. This can be done either by checking how well a system conforms to a set of design characteristics, or by generating abstractions of a system and then considering how the goals relate to the abstraction.

When checking the conformance to design in source code, developers may either look for posited structure or pattern compliance. Two examples of such techniques are Software Reflexion Models [51] and PatternLint [63]. Software Reflexion Models (RM tool) allow the developer to check the conformance of system code to intended structural properties. PatternLint checks whether a pattern is implemented correctly. Information about the code, such as calls and variable dependencies, are stored in a fact base. Facts describing structural features of patterns are also included. A developer may then compare the system against a pattern description using a series of conformance and non-conformance rules.

Using visualization techniques, such as [59, 31], developers can extrapolate structural and behavioral information from the source code itself. Richner et al. [59], for instance, provide a mechanism for customized generation of views of the source code extracted from static and dynamic information about the source. Pattern mining techniques, such as [56, 34, 42], help a developer search for and recognize design patterns used in the source code comprising a system. The Pat system [56], for instance, uses Prolog and a commercial CASE tool to locate instances of structural design patterns in source code.

Neither conformance, nor extrapolation techniques elucidate the rationale behind the implementation of a code base. Mechanisms related to design patterns come part of the way to relating design rationale information, since they identify structures which can be compared against design pattern descriptions. However, these mechanisms do not help a developer determine which variants of a design
pattern were used in a particular code base, understand why the design pattern was applied, or map goals from the design pattern down to the implementation.

1.2.3 Trace Design Concepts to Code

Several approaches have been proposed to help developers trace the design rationale pertinent to a code base.

Knowledge-based techniques provide the developer with the ability to search for abstract goals and non-functional requirements in a software system. Two examples of such techniques are LaSSIE [17], where the knowledge base is built before the tool is used, and DESIRE [8], where knowledge is incrementally included in a domain model as developers use the tool. The effort required to apply these techniques is non-trivial; it is unclear if the work is tractable within the context of a software evolution task. Additionally, LaSSIE does not allow for encoding of information such as “why is this action performed here” or “is this operation involved with more than one feature”. This limitation reduces a developer’s ability to see how certain kinds of design goals are carried out in the code, and also to determine why the code exists as it does: the rationale behind the code.

Other approaches that allow exploration of existing artifacts include hypertext style mechanisms. Hypertext allows non-linear navigation of documentation. Some hypertext systems, such as CHIME [18] enable linking within source code to allow browsing from variable definitions to uses, and from a caller to a callee. Some, such as SODOS [30] offer linkage between non-source documents. The mechanisms that go farthest in terms of relating rationale to code are those techniques, such as SLEUTH [22], that link design documentation to source. In SLEUTH, links within the documents are generated when the author specifies filters indicated by regular expressions. The links can point to files, or to anchors within documents. The within-document anchors must be inserted manually. SLEUTH operates on documentation that has been designed for hyper-linkage. Many of the artifacts that
describe software systems, including design patterns, do not satisfy this constraint. Additionally, in SLEUTH, links to code are at the file level. This limits the granularity at which concerns in code can be pinpointed.

Antoniol et al. used text analysis to form links from documentation to source code [1]. They applied a language model in which probabilities are assigned to every string of words taken from a prescribed vocabulary. In their approach, text and code are analyzed. The design documentation text is used to estimate a language model. The code is analyzed and translated into an intermediate and interpretable form. They then assign probabilities that indicate whether a portion of code resembles a document. They applied the technique by associating manual pages with the source code they describe. This approach is useful in terms of determining which documents relate to which portions of a code base. However, it does not enable a developer to deconstruct a document to identify portions of text related to particular design goals or rationale, or to determine which portions of code are relevant to those goals.

1.3 Overview of the DPRG Approach and Tool

This dissertation presents the Design Pattern Rationale Graph (DPRG). A DPRG is intended to aid a developer when changing a system. A DPRG graphically represents the text of a design pattern as it appears in its original description, and links that representation to source. The representation is navigable through use of the mechanisms provided by the DPRG tool, enabling a developer to browse from design goals described in a design pattern's text down to the code that implements those goals. This overview outlines how a developer may reason about the code in terms of the design goals that affect it, and that are affected by it.

1.3.1 DPRG Structure

As is shown in Figure 1.1, the DPRG is composed of three levels: the pattern-level, the source-level, and the link-level.
The pattern-level provides a graphical representation of the text of a design pattern. The DPRG represents design entities and concepts from the design pattern as nodes. Edges between the nodes show relationships between the concepts and entities. Sentences from the design pattern are depicted in the pattern-level as chains of nodes.

The source-level graph represents a particular code base. Source-level nodes represent source code entities, such as classes and methods, from a particular code base. Edges between the nodes express relationships between the entities, such as calls and has.

The link-level graph contains edges that show correspondence from the pattern-level to the source-level. The link-level overlaps both the pattern-level and source-level; it contains pattern-level and source-level nodes that correspond, and contains edges from the pattern-level and source-level if both the source and destination nodes of those edges are in the link-level.

1.3.2 An Introduction to the DPRG Tool

A developer navigates the structure of a DPRG by performing queries which result in graphical views. Using these queries, the developer can navigate "down" from portions of the design to the code, and "up" from the code to relevant portions of the design.
Figure 1.2: An abstract depiction of the OBSERVER pattern as described in [23]. Grey areas represent places where design options are offered.

To illustrate the key features of the DPRG tool we present a sketch of its use by an engineer in a representative source code investigation task. In this scenario, a developer has been asked to improve the efficiency of a drawing editor that is based on the JHotDraw object-oriented framework. The design of JHotDraw relies on several design patterns [23], including the OBSERVER pattern [23].

Figure 1.2 depicts an abstract view of the OBSERVER pattern as described in the text [23]. This pattern presents a situation in which a set of objects, called observers, depend on a state of another object, called the subject. This pattern could also be referred to as a publish-subscribe scenario. In the OBSERVER pattern, subjects are aware of their observers. Observers attach themselves to a subject, and are then notified of changes. The notification causes the observer to update its own state. Several design options are presented in the pattern, including who triggers the notification of observers, how a subject keeps track of its observers, and how often notification should occur.

We pick up the scenario after the developer has decided to investigate the
efficiency of an implementation of the OBSERVER pattern in a particular portion of the JHotDraw code base, and after the developer has created a DPRG for this pattern and code.

The developer uses two kinds of query operations to select relevant portions of the DPRG to view: regular expression searching, and node expansion. Regular expression searching enables the developer to view portions of the graph containing that expression. By expanding nodes relating to concepts or entities, the developer can view all portions of the graph relating to those concepts or entities. Using these two kinds of queries, the developer can navigate through, and browse from a conceptual level down to the relevant portions of source code.

Initially, the developer starts a new, blank, view of the DPRG. Then the developer performs queries and expansions to learn more about efficiency in the OBSERVER pattern.

First Investigation

The developer begins by searching for the regular expression efficien to capture concepts involved with both efficiency and inefficiency, such as efficient, efficiency, inefficient, and inefficiency. The result of this query is the pattern level portion of the DPRG shown in Figure 1.3. Viewing these results, the developer sees that the “pull model may be inefficient”. Referring to Figure 1.2, the pull model is related to the design choice “how to tell” in Figure 1.2. It is a way to design the participants in the OBSERVER pattern such that the subject sends very little notification information to the observer.

The developer then wishes to see how the pattern-level subject node relates to the JHotDraw code base. The developer can perform a downward node expansion on the subject node. Added to the view is the link-level shown in Figure 1.3, in which the subject node has been expanded at the link-level. The nodes added include the source-level node AbstractFigure, which plays the role of a subject in
Figure 1.3: Three queries on the OBSERVER/JHotDraw DPRG: a pattern-level regular expression query for efficiency, a link-level expansion of subject, and a source-level expansion of PolyLineFigure.
the JHotDraw code base. Also added are link-level nodes that neighbour the subject and AbstractFigure nodes.

To view more of the source-level context for AbstractFigure, the developer chooses to expand PolyLineFigure at the source-level. Two nodes are added to the view as a result of this action: a constructor method, and the LineConnection class. This expansion is depicted in the portion of Figure 1.3 labeled Source-Level.

Second Investigation

The developer then decides to expand the pull model node at the pattern-level for more information about the concept. All the sentence chains containing the phrase pull model are then added to the view. This is partially depicted in Figure 1.4. The added chains include: “the pull model sends minimal notification”, and the pull model “emphasizes the ignorance of the subject”. Another chain includes an alternate strategy: the push model.

As a result of viewing these queries, the developer determines that efficiency might be improved by applying the push model. In contrast to the pull model, in the push model the subject sends itself as a parameter when it notifies the observer of a change in its state. From the phrase “the pull model sends minimal notification”, the developer infers that changing the implementation to follow the push model would likely require modifications to the notification mechanism. Thus, the developer decides to perform a regular expression query for notif. The resulting sentences that are displayed come from the pattern-level of the DPRG, and include a node called notify method. To see how this node relates to the code base of interest, the developer performs a downward expansion of the notify method node. This allows the developer to see how the notify method corresponds to portions of code. The developer is then shown the invalidate method at the link-level, as well as the figureInvalidated method connected to the invalidate method by a calls relationship. The developer is also shown the other pattern-level nodes connected to the pattern-level notify
Figure 1.4: Results of a regular expression query for *efficient* followed by an expansion of the *pull model* node. This figure is provided to illustrate the results of this query; the graph is difficult to read unless shown on a screen. The added chains include the phrases: "the pull model sends minimal notification", and the pull model "emphasizes the ignorance of the subject". Another chain includes an alternate strategy: the *push model*. 
Figure 1.5: Partial results of a regular expression query for `notif`, and expansion of the `notify method` at the link-level. In this figure, links from pattern-level to source-level nodes are shown in bold. The link-level expansion (shown in the box at the bottom of the figure) shows that the `notify method` is linked to the `invalidate` method, and also reveals that the `update` method is linked to the `figureInvalidated` method. Due to the size of this graph, it is not clearly legible unless shown on a screen.
**method** in the link-level. Partial results of the regular expression query and the downward expansion are shown in Figure 1.5.

By using both regular expression and node expansion queries, the developer is able to investigate which parts of a pattern are related to a design concept such as efficiency, and to then view how those concepts are implemented in the source. The developer is now able to see how efficiency involves the push and pull model designs of the **OBSERVER** pattern, and has a starting point for where to look in the JHotDraw code to affect them.

To learn the design goals motivating a portion of the implementation, the developer can expand "upward" from a source-level node by expanding the source-level node at the link-level, revealing which design pattern element or elements correspond to the source-level node. Then the developer can use pattern-level exploration to follow through to design goals.

The DPRG can help the developer propose and investigate different alternatives for improving the efficiency of JHotDraw.

### 1.4 Organization of the Dissertation

Chapter 2 describes two preliminary studies which motivate and provide validation for the assumptions which underlie this work. In Chapters 3 and 4, we describe the DPRG model and DPRG tool, in which we refer to the example outlined above as a basis for description. Chapter 5 describes the validation of the thesis claim, providing evidence from four studies: two which address the main claims of the thesis, and two which address supporting claims. Chapter 6 discusses open issues related to the main aspects of the DPRG approach and tool. Chapter 7 covers related work. Finally, in Chapter 8 we review the thesis of this research, describe the contributions of this work, and discuss future avenues of research.
Chapter 2

Preliminary Studies

The thesis of this research makes two assumptions: First, developers encounter and deal with crosscutting concerns in code bases. Second, it can be difficult for developers to find or recall design context (design goals that motivate the code) or detail (how the design goals are carried out) related to specific concepts in design patterns.

To obtain empirical evidence of these assumptions we performed two studies. The first study was an inquisitive study that focused on the first assumption. The second study was a comparative experiment that provided data about the second assumption. The experiment is presented in its entirety in Chapter 5.

The validation of these assumptions both substantiates our observation of the problems faced by developers, and motivates and frames the work contained in the subsequent chapters.

Introduction to the Preliminary Studies

Although there has been empirical study of how developers approach the task of understanding a code base, there has been little empirical work on whether and how developers encounter crosscutting concerns in their code bases.

Empirical study of how software developers approach source code has largely
focused on the mental and cognitive models built by programmers when understanding code, and on the work practices used by the developers.

The empirical study of the mental and cognitive approaches has considered four different approaches: top-down approaches [33, 65], where a programmer begins with understanding a code base in a general way and then refines their understanding; bottom-up approaches [62, 55], where programmers form higher-level abstractions about a code base through reading the code; knowledge-based [45] approaches in which a developer forms a mental model of a code base through program analysis combined with their own domain knowledge; and integrated approach [73], which is a combination of the three other approaches.

The empirical study of work practices has considered how tools can aid the understanding process. Storey and colleagues [69] describe a set of cognitive issues to be considered when designing a software exploration tool. They examined a number of cognitive models of program comprehension, and gave examples of how the use of these models could be facilitated by tool support. Singer and Lethbridge [44] conducted a set of field studies in which they analyzed the work practices of developers as a means of directing tool development. They collected information in four ways: through tool usage statistics, through a questionnaire, through group studies, and by shadowing or observing a developer for an extended period of time. Based on this data, they concluded that programmers need support when searching code bases in terms of being able to store the results of their search and for recalling their movement through code bases.

These empirical approaches place emphasis on the work practices used and the types of mental and cognitive models built by programmers while understanding code. This work describes the methods developers use in understanding how particular algorithms or structures are distributed throughout code. However, the broader question is still unanswered: Whether developers encounter crosscutting concerns in their code, and if they do, then with what effect?
The first study we describe in this chapter is the only study to date that has focused on whether developers encounter crosscutting concepts, how they identify these concepts if they occur, and the types of actions they take to manage these concepts.

The second assumption on which this thesis is based is that it can be difficult for developers to find or recall design context (design goals that motivate the code) or detail (how the design goals are carried out) related to specific concepts in design patterns.

There has also been limited empirical work on the how well developers are able to understand design patterns. Studies on patterns have focused mainly on their application. For instance, Prechelt et al [56] conducted a study which compared the quality of work done by developers on a system in which design pattern related comments were included, with work done by developers on the same system with the comments removed. They found that developers viewing code containing comments pertaining to design patterns were able to produce better results than those viewing code without design pattern comments. Another study [57] examined the effect of design patterns on designer communication, but did not comment on how well developers are able to understand a particular pattern as indicated by their communication.

The second study we describe in this chapter is the only study to date that has focused on how well developers were able to report design concepts within a particular design pattern.

2.1 Study: Developers and Concerns

Our goal for this study was to determine whether developers encounter crosscutting concerns in their code, and if they do, how they identify them, and deal with them [5].

To fully investigate the experiences of programmers, we chose to use an
exploratory study rather than a study intended to verify a particular hypothesis. To that end, we constructed an inquisitive study adapting a portion of the method used by Singer et al [44]. In our study, participants were interviewed about their experiences performing software change tasks.

This study involved eight participants with a broad range of backgrounds: four were practicing software engineers from Siemens AG and had years of programming experience in an industrial setting; four were graduate students from the University of British Columbia with a range of programming experience.

To participate in the study, we required that a participant be working on, or have recently worked on, a program change task to a system for which they were neither an initial nor a principal developer. We set this constraint to ensure that participants would have to investigate the scope of the change since they had limited prior knowledge of the code base. Each participant in the study was working on a separate system.

We asked participants whether they were familiar with aspect-oriented programming [36] (AOP), a programming style that seeks to reduce crosscutting concerns, of particular types, in source code. Only two of the participants had exposure to AOP. One of these two participants was actively applying AOP ideas in the change task with which they were involved during the study, and the other had experience working with an aspect-oriented language. The rest of the participants had no knowledge of AOP at the start of the study.

Before commencing the study, participants were asked to provide the interviewer with a copy of the code they were working on to serve as a reference.

2.1.1 Design

We organized the study as a series of interviews: each participant was interviewed three times over a period of three weeks, with each interview lasting up to an hour. The same interviewer conducted all interviews.
General guidelines to focus the interviews were prepared in advance. The specific questions that were asked depended upon the flow of conversation. Our goal was to determine four different pieces of information during each interview:

1. the program change task of the participant,
2. the approach of the participant to the task,
3. the approach used by the participant to determine which pieces of code needed to change, and
4. whether the participant thought that the change was difficult to make and if so, why it was difficult.

To help focus the discussion, participants were asked to identify the portions of code that had, and were, being changed. To keep track of these locations, we annotated the interviewer's copy of the source files.

All the interviews were audio taped and later transcribed.

Questioning Convention

A primary purpose of the study was to determine the kinds of crosscutting concerns that developers must consider in existing code bases. Rather than ask the participants directly about these concerns, we asked them questions about the change task on which they were working, and attempted to glean concerns from their responses.

We took this approach for three reasons. First, most of the participants had never thought about crosscutting concerns. When we attempted to pose questions that directly asked about concerns, the participants were unable to understand the meaning of our questions. Second, there was a danger that the participants who did have some knowledge of this area would simply respond with crosscutting concerns used as examples for AOP languages, such as tracing, debugging, or distribution.
Such a response might have hidden more task-related concerns. Third, when programmers are heavily involved in the details associated with a task, it takes time to ease them into coarser-grained thinking about their problem. Asking participants questions that they could answer readily from their own experience facilitated the gathering of data.

At the beginning of an interview, participants tended to talk about their change task in a detailed way. For example, one participant provided in-depth information about specific data structures used in the application. Typically, by the end of an interview, participants began to talk about their task at a more conceptual level. This shift in the level of detail enabled participants to consider higher-level questions, such as labels that they might use to describe the code they were examining, or methods that they had used to find the relevant portions of source for their task. The more conceptual level of thinking about the task enabled the interviewer to ask participants to think, between interviews, about the following question: If you could have any view of the code, what view would have helped you perform this task? This question was intended to help identify the portions of code that the participant would like to see modularized. To help make the question concrete, the interviewer provided sample answers, such as “all the code pertaining to the database system”, or “all the code related to printing”.

Method of Qualitative Analysis

To analyze the data, we examined the transcripts of the interview sessions and the annotated source code.

Our examination involved three passes over the transcripts. First, we perused the transcribed interviews to understand the range of responses. Second, we categorized the responses in terms of how the participants described the change they were performing, and what difficulties they reported to have encountered while working on the change. Finally, we examined the responses for commonalities.
We also examined the annotated source code, looking at the form of the statements highlighted. We looked for commonalities in terms of syntax, semantics, and function. We also examined the code to determine whether the changes themselves could be characterized as belonging to a particular crosscutting concern.

This analysis was primarily performed by the investigator.

2.1.2 Results

In this section we describe the results of the inquisitive study. We begin by providing the results of the qualitative analysis of the study data, and then discuss study validity. We end by discussing the implications of the results on tracing crosscutting concerns.

Qualitative Analysis

Most participants described their change task from two perspectives: a structural perspective, and an emergent obstacle-based perspective. Almost every participant at some point in an interview used the phrase: “Everything was going fine until . . .”. We describe each of these two perspectives and then describe the results of an analysis on the change and obstacle code.

Straightforward Structural Perspective

Each participant began by providing detailed descriptions of the application problem domain and of the change. They described the field in which they were working, how their application fit into that field, and how their change fit into the application.

The participants’ initial descriptions of their change task were in terms of easily identifiable structure in the code. Specifically, most participants described their change in terms of a particular data structure or a particular module in the code, such as “I was changing the components of a data structure”, or “I was changing the methods related to the user interface”. Describing the change in this
way was straightforward, even though the code was often spread throughout the
code base. The programmers could understand the purpose of the code and its
context within the structure of the application. They could point out portions of
the code that corresponded to their change.

Only Participant one described crosscutting code as the target of the change.
This participant was working in the area of aspect-oriented programming.

Non-straightforward Obstacle Perspective

After participants had described their change task, and after they had pointed out
the locations in the code that they had to change, we asked them to consider if these
were the only portions of code that had to change to complete the task. Invariably,
they said “no”. It was at this point that the participants revealed a set of obstacles
that they had encountered when making the change.

Obstacles comprised portions of code that were relevant to the task but that
also affected an underlying crosscutting concern; this code was at the intersection of
the core change and the broader concern. For example, Participant eight was adding
a feature to the system and had to ensure that the change was consistent with be­
havioural conventions. To make the change, the participant had to overcome the
obstacles and to try to understand the entire underlying concern, the behavioural
conventions, that led to the presence of that portion of code. Since that underly­ing
concern was neither well-modularized nor well-documented, it was difficult to
conceptualize and to reason about.

The participants used three strategies to cope with the obstacles:

• Change: Alter the concern code to enable the change task.

• Within: Understand, but do not change, the underlying concern associated
  with the obstacle sufficiently to make the change work within the concern.

• Around: Completely alter the change task to account for the concern without
Table 2.1: Program Change Tasks and Strategies

<table>
<thead>
<tr>
<th>Participant</th>
<th>Structural View</th>
<th>Obstacle View</th>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Moving particular computation to an aspect-like module</td>
<td>Synchronization, Performance</td>
<td>Within</td>
</tr>
<tr>
<td>2</td>
<td>Tailoring a matching algorithm for a specific purpose</td>
<td>Memory allocation</td>
<td>Change</td>
</tr>
<tr>
<td>3</td>
<td>Changing matrix calculation</td>
<td>Memory allocation</td>
<td>Around</td>
</tr>
<tr>
<td>4</td>
<td>Changing table representation</td>
<td>Implicit assumptions about data structure representations</td>
<td>Around</td>
</tr>
<tr>
<td>5</td>
<td>Changing packaging of user interface mechanism</td>
<td>Distribution, Tracing</td>
<td>Within</td>
</tr>
<tr>
<td>6</td>
<td>Changing the mathematical model applied</td>
<td>Security issues, Communication protocols, Hardware platform dependencies</td>
<td>Within</td>
</tr>
<tr>
<td>7</td>
<td>Changing printing look and feel</td>
<td>User interface consistency, Printing speed</td>
<td>Change</td>
</tr>
<tr>
<td>8</td>
<td>Adding cancellation notification to an existing system</td>
<td>Multithreading, Behavioural consistency</td>
<td>Within</td>
</tr>
</tbody>
</table>

understanding it.

Table 2.1 summarizes the program change tasks for each participant, the obstacles each encountered and the strategy each employed.

Change Strategy Participants two and seven used the first strategy: They changed the relevant portions of the crosscutting concern to suit the change. For Participant seven, this approach was facilitated by the fact that the changes were at the user-interface level, and thus were more visible during testing. Participant two's changes are discussed in more detail in Section 2.1.2.
Within Strategy  Participants one, five, six and eight used the second strategy: They worked hard to understand the effect of their code on the crosscutting concern that presented an obstacle to their change, and they worked within the conventions of the concern. Participant eight had to perform considerable testing to ensure the obstacle had been dealt with appropriately.

Around Strategy  Participants three and four used the third strategy: They each worked around the obstacle. They significantly re-thought their original approach to their change task because they could neither adequately understand the obstacles, nor address the concern. Participant four, for example, ran into memory allocation problems after making what should have been a simple change to a table representation. After failed attempts to understand how the change affected the memory allocation for the application, a work-around was devised to trick the memory allocation portions of the source into thinking that the change had not been made.

Code Perspective

By examining the code associated with the changes and associated with the participants' comments, we learned more about how participants addressed the obstacles they faced. Our examination focused on the obstacle points; the locations at which the change task intersected the crosscutting concern. We discovered that there were certain patterns of interaction between the concern and the change code, and we determined that there was a correspondence between the patterns and the strategy the participant chose to address the obstacle.

Change Strategy  Code associated with participants who chose the first strategy, the change strategy, had a structural intersection point. From the code related to the change, participants could identify certain structures as obstacles to their change task. Such structures included types, objects and computation directly related to those structures. Figure 2.1-A depicts this situation. These obstacle points,
shown as black boxes, provided enough information about the broader concern to lead the participant along the outward reasoning arrow to the points of change, located in the broader concern shown in a dotted box. This situation was particularly true for Participant two. For this participant, the obstacle points were easily identifiable by the type of the data structure affected. Participant two was able to extrapolate that all functionality of a certain kind involving a particular type would have to be altered. It was then straightforward, though tedious, to make the changes.

**Within Strategy** Code associated with participants who chose the second strategy, the within strategy, followed a behavioural pattern. Participant eight worked within computational conventions, and Participant one had to work within a particular synchronization policy. The intersection of the change code and the behavioural concern code could not be as easily assessed as for the structural case above. As is shown in Figure 2.1-B, the obstacle points were implied. Comments alerted each of the participants to the presence of the obstacle, and gave them clues as to the existence and nature of the broader concern. Based on the comments, these participants had to examine the broader concern to understand the conventions of
Figure 2.2: Code Alterations show Inward Reasoning

The participants then had to reason inward about how to change the core code to work within the broader concern. Their analysis techniques were ad hoc, and it was difficult for them to describe their approach. Essentially, they reported that they had to gain a general understanding of the code base in order to work within the concerns. Once they gained this understanding, they were able to identify portions of code that would allow them to reason inward about their specific change task.

Figure 2.2 shows the inward reasoning, and resultant code, used by Participant one. This participant was moving pre-fetching functionality within operating system code into a separate “aspect-like” module. Specifically, the participant wanted to migrate the circled code in the black box on the left of Figure 2.2 to the pre-fetching module on the right. Based on previous knowledge, and on comments
in the code, the participant knew that this change would affect the goal of synchro-
nization in the system. Relevant synchronization code, shown in boxes A1 and B1,
was identified by traversing up the call chain and pinpointing locking and unlocking
code that could affect the code of interest. The developer then had to reason inward
from the synchronization concern to the core change. Synchronization code similar
to that in boxes A1 and B1 had to be included in the new pre-fetching module
(boxes A2 and B2) even though the code was not directly related to the core of the
change. The inclusion of this code ensured that the locking invariants encoded in
the synchronization concern were maintained.

In all cases, participants were unable to cleanly determine when they had
addressed all of the code affected by their change; they were unable to identify
all the code carrying out relevant design goals. Our examination of their code
yielded limited similarities about the nature of the concern code. In particular, for
participant eight, the concern conventions could be gleaned by scanning for instances
of a particular sequence of calls. When asked, Participant one reported that this
“sequence of calls” analysis might have been helpful. Participant one might also have
been helped by information about a pattern of access to particular data structures.

Around Strategy  Obstacle code encountered by participants who chose the
third strategy, the around strategy, was dense. The code made ambiguous use of
assumptions from around the code base and was thus subtle and difficult to reason
about. Originally, these participants had wanted to change the relevant portions,
rather than to avoid the code. However, when the change approach became too
onerous, the participants were forced to work around both the obstacle and the
concern code. It was typically unclear why particular data structures were altered
in particular ways, and why the ordering of certain computations was important.
For instance, the obstacle code encountered by Participant four assumed that a data
structure of a certain number of bytes (16) would be used. This number was relied
upon heavily in the computations for allocating memory, but was never indicated
if (mem != NULL) {
/* successful allocation; slice memory */
  ptruint offset;
  unique->memused +=
    (DD_MEM_CHUNK + 1) * sizeof(DdNode);
  mem[0] = (DdNodePtr) unique->memoryList;
  unique->memoryList = mem;
  offset =
    (ptruint) mem & (sizeof(DdNode) - 1);
  mem += (sizeof(DdNode)
    - offset) / sizeof(DdNodePtr);
...
asked of participants focused on the changes being performed rather than on the crosscutting concerns encountered. Despite the differences in tasks and systems, similarities emerged in the form of the crosscutting code involved, and in the strategies used by the participants to cope with the crosscutting concerns. The presence of these similarities in the context of the differences between participants, systems, and tasks increases our confidence that the results are indicative of real software developments and that the results may generalize.

Two limitations of our study are the small number of cases considered, and the short amount of time for which we tracked the progress of the developers.

Implications for Tracing Design Goals and Rationale

This study shows that regardless of task, programming language, or language type, all programmers likely meet with obstacles when making changes associated with a task. By analyzing both the participants’ responses and their code, we determined that these obstacles appear at the intersection of a portion of code with a crosscutting concern.

Depending on the nature of the code at the point of intersection, developers choose different strategies for dealing with the obstacles. Two cases, in particular, point to the need for a better understanding of design goals behind code. Developers who chose to use the *within* strategy needed to better understand how their code of interest related to the crosscutting concern encountered. They may have benefited from context-specific information about design goals (rationale) associated with portions of code. For instance, Participant one did not need to understand the entire synchronization design for the system. This participant needed a view of the relevant parts of the synchronization scheme pertaining to the code being changed. Other developers chose to work around their obstacles. These developers were forced to do this based on a lack of information about how their code related to others in terms of the design goals which motivated them.
2.2 Study: Detail and Context Reporting

To better understand how well developers are able to find and recall design information from design patterns documentation, we conducted a small study in which we asked four developers to read and answer questions about a design pattern.

2.2.1 Design

All participants were asked questions about one of two design patterns. We recorded their responses and compared them against the information available in the pattern text. This convention was adapted from a portion of the methodology of Prechelt et al. [56].

Two participants were software developers from Siemens AG, and two were graduate students from the University of British Columbia. The Siemens participants worked with the REACTOR pattern [61], and the UBC participants worked with the VISITOR pattern [23].

Patterns Used

We used two patterns to help reduce the likelihood that a problem in understanding the pattern was related to the way in which the pattern was written, or to the questions we asked. The two patterns we used have different authors and are of differing size: the VISITOR pattern (at ten pages) is short but subtle; the REACTOR pattern is (at 22 pages) is longer and more detailed.

The VISITOR pattern describes a design solution for determining which method to execute based on the types of both the caller and the callee. The main participants in the VISITOR pattern are the visitor, which stores the type of the caller, and the element, which is the callee. When an element method is called, the visitor is passed as an argument. The element then uses the visitor object to execute the appropriate operation.
The REACTOR pattern describes a solution in which event-driven applications handle requests for service sent to an application by one or more clients. The participants in this pattern are the handler, which handles incoming requests, the demultiplexer, which indicates which handles can have operations invoked on them, and the initiation dispatcher, which allows registration, removal and dispatch of event handler objects.

Participants

We kept the skill set of participants working with each pattern as similar as possible. Both of the participants in the REACTOR experiment were non-native English speakers, with similar experience in reading and writing English. Each participant possessed the equivalent of a Bachelor's degree in Computer Science, and had at least one year of experience working with Java in an industrial setting. Each participant was screened to ensure a certain level of exposure to design patterns, but no exposure to the REACTOR pattern.

The participants working with the VISITOR pattern were PhD students of Computer Science at the University of British Columbia. Both participants had academic experience in software development, but no experience developing large systems. Neither of the participants had previous exposure to design patterns. Each participant was a native English speaker.

Experimental Set-up

In each trial, a participant was given the same set amount of time to read a hard-copy of the assigned design pattern. The participants were asked to read the text for the time given, and were told that they would be required to answer questions based on their understanding of the pattern. Participants in the VISITOR trials were given an hour to read the pattern, and those in the REACTOR trials were allowed two hours. These times were chosen based on the length and complexity of each
Each participant who read the VISITOR pattern was asked to answer, as fully as possible, three questions. For each question we note how many places in the design pattern text were relevant for a correct response.

1. What allows the VISITOR to directly access the concrete element? (Two places in the text were relevant)

2. How is it determined which operation is executed? (Four places in the text were relevant)

3. What is the sequence of events that occur in the VISITOR pattern? (One place in the text was sufficient for complete information)

For the REACTOR study at Siemens, the experimenter was not on-site. Instead, the experimenter interacted with each participant over the phone and over the web. Each participant was asked the following questions pertaining to the REACTOR pattern:

1. What does the logging handler register with, and what does it register for? (Six places in the text were relevant)

2. About what does the synchronous event demultiplexer notify the initiation dispatcher? (Three places in the text were relevant)

3. What happens after a connection request arrives? (Five places in the text were relevant)

These questions could be answered based solely on the information in the text of the pattern. Participants were given 20 minutes to answer each question, and were allowed to re-read the pattern within the allotted time. Participants were given the patterns to read before being asked to consider the questions in order to simulate an typical design act in which a developer reads a pattern that may
be relevant to a particular design issue and then, some time later, refines their understanding possibly by re-reading the pattern. We simulated the re-reading portion of this activity by allowing control group participants to revisit the text of the design pattern in order to answer the questions.

**Evaluation Questions**

After the participants had responded to the pattern-specific questions, they were asked follow-up questions.

Participants were asked about their confidence in their answers to the pattern questions, how they used the pattern text to reach their answers, and from where in the pattern text they drew their answers.

**2.2.2 Results**

The results of our study showed that while developers given the design patterns were able to answer questions about the crosscutting elements from a design pattern, they did not do so completely.

**Detail and Completeness of responses**

For both patterns, participants typically responded with high-level conceptual information. Participants reported only limited design detail in answering questions, and only limited context.

The first question in the REACTOR trials, for instance, dealt with a specific example from the pattern. The participants tended to answer about the general case, rather than about the situation described specifically in the example. Although their answers demonstrated that they understood the general intent, they missed stating the precise type of events registered for by the logging handler used in the example, the type of event handler it registered, and details about the entity with which it registered. The developers were unable to report about the process of the
registration for events.

In the third question of the REACTOR trials, the participants were asked to explain what happens after a connection request arrives. Even when taken together, the Siemens participants missed many design details. Among them, the passive establishment of a `sock_stream` object, and the invocation of the synchronously demultiplexing `select` call.

We realize that developers intending to apply these patterns may have probed more deeply into them, and performed more detailed analysis of the text to attain a more detailed understanding. However, the time given each of the participants to initially read, and then to revisit the text was sufficient in order to answer these questions in an in-depth manner. This observation indicates that developers do have trouble reporting design detail and context with relation to concepts in a design pattern. The next observation suggests reasons for this.

**Hesitation to Explore**

Each of the participants took less than five minutes to answer each of the questions. When we asked them why they did not take more time to re-read the pattern to provide detail, two of the control participants said they did not know it was required, implying that they would not voluntarily do so, and the other two explicitly said they did not feel they would get anything more out of “flipping through” the text.

For example, one participant in the VISITOR trials, responded incorrectly to Question 1. When asked why he did not look in more detail for the relevant parts of the pattern, one VISITOR participant said:

[it] seemed familiar, but I didn't think I could flip back and find it. I did kind of hesitate with the text going [sic] “do I remember at all where that was, or am I going to have to re-read the whole thing?”, and then decided I had a pretty good idea where [it] appeared.

Another indication of the hesitation to explore was the disinclination of the
Table 2.2: Confidence and Correctness of Participant Responses

<table>
<thead>
<tr>
<th>Participant</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>REACTOR Participants</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>confident, generally correct</td>
<td>confident, generally correct</td>
<td>confident, generally correct</td>
</tr>
<tr>
<td>2</td>
<td>less confident, generally correct</td>
<td>less confident, generally correct</td>
<td>less confident, generally correct</td>
</tr>
<tr>
<td><strong>VISITOR Participants</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>less confident, incorrect</td>
<td>confident, incomplete</td>
<td>confident, incomplete</td>
</tr>
<tr>
<td>2</td>
<td>less confident, incorrect</td>
<td>confident, more complete</td>
<td>confident, incomplete</td>
</tr>
</tbody>
</table>

participants to modify their original answers. All participants were asked to look through the pattern and to report the source of their answers. They all used this perusal to support their original answers, even when those answers were incorrect as happened with two VISITOR participants. This was the case even in the VISITOR pattern Question 3, in which only one portion of the pattern description was needed to correctly answer the question. Both participants reviewed this single location, and reported no change to their original answers.

Level of confidence in answers

The levels of confidence, and degrees of correctness of the participants’ responses are shown in Table 2.2. Although all participants reported that they could not be fully confident about the completeness of their answers, they were confident about the correctness of their answers. One REACTOR participant reported little confidence about the correctness of all of his answers. The two VISITOR participants expressed
complete confidence in their answers to questions two and three, but less confidence about question one, although they both strongly believed that they were partially correct. The other REACTOR participant felt confident about all of his answers. To see if they were correct in their levels of confidence, we examined their answers.

As mentioned before, the REACTOR participants both answered all the questions mostly correctly, while missing details of the answers.

The participants working with the VISITOR pattern both rightly lacked confidence on question one, which they both answered incorrectly. For question two, about which they were both highly confident, they answered incompletely, forgetting that it is both the type of the visitor and the type of the concrete element that determine which visit operation is eventually called. One said that it was only the type of the visitor; the other answered more completely, responding that only the type of the concrete element determines the operation. The second is more complete since the participant may have considered that the relevance of the visitor was implied by the question. In question three, only one participant was able to give details about the invocation of the visit and accept operations; neither recalled how the accept operation was called (even though one was referring to the interaction diagram shown in the text, page 335, [23]).

**Study Limitations**

The format of the study has several drawbacks, including the small number of participants, and the small number of patterns.

The small number of users and design patterns in our study affects the generalizability of our results. We chose to limit the number of users and patterns because we wished to gain only an indication of the completeness and confidence with which developers were able to report design detail and context.

We believe our results provide this indication because we varied the background of the participants, including both experienced software engineers and grad-
uate students, and because we selected patterns that differed in authorship, style, and length. We chose to give participants with similar backgrounds the same pattern to examine so as to reduce variability of developer experience within the pattern groups.

Summary of Results

We noted that participants did not report design detail and context with relation to elements of the design pattern.

We also noted that while participants were not necessarily confident in their responses, they chose not to re-read the design pattern to solidify their understanding. This hesitation limits the usefulness of design pattern texts.

Though this study was very small, we believe it provided interesting indications of developers’ abilities in terms of reading and understanding design patterns texts. Design patterns are not always understood, hence not implemented correctly. Although all the information pertaining to the questions was present in the text of the design patterns, the participants did not navigate through the text to report all the information pertaining to these elements. Perhaps, had we insisted that they do this, they would have provided correct answers, however, it is not at all likely that this would have produced realistic results. Taking software developers’ time constraints into account, it is more likely that developers would read the pattern in the same way as these participants did, and check their knowledge to a similar degree. In a study considering industrial cases presented in Section 5.2, we saw similar approaches and depth in reading of design patterns. Additionally, given the fact that our questions implied that we wanted complete answers, and given the participants’ persistent hesitation to re-read the pattern to solidify their responses, it is likely that had we required a more detail-oriented process this would have seemed a tortuous task. If a developer wished to understand a portion of code implementing a design pattern it is likely that this is the type of reading they would perform, and
that the responses given by the participants would somewhat accurately represent their knowledge during such an investigation.

### 2.3 Summary

In this chapter we have presented two studies that verify the underlying assumptions of this dissertation: Developers encounter crosscutting concerns while evolving software, and have difficulty finding design detail and context related to elements and concepts whose descriptions crosscut design patterns.

In terms of the first assumption, we learned that developers encounter crosscutting concerns as obstacles to straightforward change tasks. Depending on the nature of the obstacle, developers choose one of three strategies to deal with the concern: they change the broader concern code, they work within the concern, or they work around the obstacle. In the second and third strategies, developers noted that help in identifying the rationale behind the obstacle and concern code would assist them in their task.

In terms of the second assumption, we found that developers report limited detail and context when responding to questions about design entities and concepts described in a design pattern. The limited nature of their responses may in part be due to their reluctance to explore a textual design document.
Chapter 3

Design Pattern Rationale Graph Model

A Design Pattern Rationale Graph (DPRG) is intended to provide a representation of a design pattern and source code that a developer can use to elucidate design goals motivating their implementation. To do this, a DPRG serves two roles: it makes explicit the relationships between concepts in a design pattern, and it supports the linking of pattern concepts to their implementation in a software system.

The DPRG has three levels (Figure 3.1): the pattern-level, which depicts the information derived from the text of a design pattern; the source-level which depicts the entities and relationships from a particular code base; and the link-level, which describes the correspondence between the two. This chapter describes each of these levels in detail. As a basis for these descriptions, we refer to the Observer/JHotDraw DPRG from the example described in Chapter 1.

3.1 Pattern-Level

The pattern-level in a DPRG provides a graphical representation of the sentences comprising the text of a design pattern. The structure of the pattern-level of a DPRG should facilitate exploration of design goals.
To provide a structure with which a developer can perform this investigation, the pattern-level of a DPRG must satisfy several goals:

- First, it must collect information related to key concepts and entities from the pattern text such that the information would be connected in the graph. Such a centralization facilitates a developer's investigation of elements.

- Second, it must provide a literal representation of the original text rather than a summary or abstraction of the text. This ensures that readers of the representation have all of the information present in the original artifact.

- Third, it must describe sentences of the design pattern in such a way as to preserve the original description of why actions were taking place. For instance, if a sentence contains "A calls B to do C", it is necessary to show that to do C was the reason for the call from A to B.

- Fourth, it must depict sentences in a way they can be read with relative ease.

- Finally, it must depict chains of events described in the text, even when the

Figure 3.1: Refined View of the DPRG Model
events are described over several sentences.

To address these goals, we have devised two structures: the *sentence chain* and the *sequence chain*. A sentence chain depicts a sentence in the design pattern text. A sequence chain shows that a collection of sentence chains relate. This section describes the structure of a design pattern, and then details each of the structures.

### 3.1.1 Design Patterns as Pattern-Level Graphs

There is no common format for design patterns. However, according to the text *Design Patterns: Elements of Reusable Object-Oriented Software* [23] most pattern formats share four essential elements: a name, capturing the general concept of the pattern, a problem, describing when to apply the pattern, the solution, describing the elements of the design and their relationship to one another, and the consequences, which are results and tradeoffs associated with applying the pattern.

The pattern-level graph of a design pattern is formed from the text of the latter three of these elements. The pattern-level graph includes the name, the entire problem description, and the entire consequences description. The solution portion of a design pattern generally contains three components: textual description, code examples, and diagrams that depict the relationships between pattern elements. Only the textual description portion is represented in the pattern-level of a DPRG.

In the validation of this thesis we use patterns of two styles. One, from the text *Design Patterns: Elements of Reusable Object-Oriented Software* [23], called the *Gang of Four* (GoF) book after the group of four authors, and the style introduced in the text *Pattern-Oriented Software Architecture*, or POSA style. Seven sections of a GoF pattern are used in the pattern-level of the DPRGs: intent, motivation, applicability, participants, collaborations, consequences, implementation, and related patterns. Six sections of a POSA pattern are used: context, problem, structure (textual portions only), implementation, variants, and consequences.
3.1.2 Sentence Chains

To address the first four goals for the pattern-level graph, we devised the sentence chain. A sentence chain graphically represents all of the words from a sentence. Noun and verb phrases are represented as nodes, and any words connecting these phrases are placed on the edges between them.

In sentence chains, the verb phrase nodes are chained together by edges to show their order of appearance in the original sentence. This ordering helps facilitate the reading of sentence chains. The chain of verbs is augmented by nodes representing noun phrases. Noun phrase nodes are connected to their nearest preceding verb. This connection visually links a verb to its subject, object, and modifiers, as described in the original sentence, thus preserving the reasoning behind the verb.

A node representing a key concept, such as a design entity or a goal, appears only once in the graph. This organization collects information related to a concept.

Figure 3.2 depicts the four types of nodes: verb nodes (verb 1, verb 2, and verb 3), modifier nodes (modifier 1), dictionary nodes (dictionary noun 1 and 2) and common noun nodes (common noun 1 and 2). The figure also shows three edge types: subsequent node edge (black), inferred subject edge (blue) and noun modifier edge (orange). In this section we will first describe the nodes and edges in a sentence chain, and then provide a definition of the chain.

Verbs—pink ovals

Verb nodes represent a verb phrase. Verb nodes in a pattern-level graph may have multiple incoming edges and multiple outgoing edges. These edges may indicate the subject, object or modifier of a verb, or may indicate the subsequent verb in the sentence chain. Each verb node in a pattern-level graph will have edges pointing to each noun phrase node that follows it, and that precedes the next verb in the sentence chain. These edges connect a verb node to nodes representing its objects.

1A verb phrase or noun phrase may be made up of one or more words.
For instance, following the edges from the verb *may be* in the lower right hand portion of Figure 1.3, a developer learns that the *pull model* may be inefficient, that this inefficiency may be due to the fact that *Observer classes* have to ascertain some information. The developer can then follow the edges extending from the verb *must ascertain* to find further details on what the *Observer classes* must do.
Dictionary Nouns—dark blue rectangle

Noun phrases can be declared keywords by including them in a dictionary. These keywords, or _dictionary nouns_, will appear only once in the graph. All other noun and verb phrases are depicted as individual nodes in the graph. This structure relates all information regarding a dictionary noun phrase to one node. The _observer_ node in Figure 1.3 shows how the DPRG brings related parts of the pattern text together, since it is included in two sentence chains.

Common Nouns—light blue rectangle

Nouns and noun phrases that are not declared to be dictionary nouns appear once in the pattern-level graph per instance in the text. Even within a sentence chain, a _common noun_ will be displayed once for each instance in the original sentence.

Dictionary Noun Modifiers—beige rectangle

If a word or phrase modifies a dictionary noun, it appears as a beige rectangle in the graph. There will never be more than one modifier node corresponding to a dictionary noun. For each modifier node, there will be an outgoing orange edge to the modified dictionary node. Modifier nodes are connected to the verb that precedes them. An edge then links the modifier node to its corresponding dictionary node. There is no link from the preceding verb to the dictionary node.

Subsequent Node Edges—black

Subsequent node edges link the nodes of a sentence chain. These edges link verb nodes to one another, and also to the noun phrase nodes that follow them. When a noun phrase is the first phrase in a sentence, it will be linked by a black edge to the subsequent nodes.
Inferred-Subject Edges—blue

Inferred-subject edges link a verb node to the noun node which is its most likely subject. These edges extend from the noun node to the verb node immediately following it. For instance, in Figure 1.3, the dictionary noun node Registration Interface, has a blue edge extending to the verb node allow. This indicates that the phrase registration interface immediately preceded allow, and that it is most likely the subject of that verb.

Modifier Edges—orange

An orange edge links a modifier node to the dictionary node it modifies. This is depicted in Figure 1.3, where the node classes modifies the dictionary noun observer. Although the phrase that generated this pair of nodes and the connecting edge is read “observer classes”, the representation in the pattern-level is read “classes of observer”. This configuration is necessary since modifiers always point to the dictionary noun they modify, regardless of how the original phrase appears.

Description of a Sentence Chain

As previously described, a sentence chain is essentially a chain of verbs, connected to dictionary nouns, common nouns, and dictionary noun modifiers, referred to as modifiers. A sentence chain has two non-terminal node types: verbs, and modifiers. Verb nodes are non-terminal since they form the basic structure of the chain: Verb nodes link to dictionary noun nodes, modifier nodes and common noun nodes in a sentence. Modifier nodes are non-terminal since they have an edge extending to the dictionary node they modify. A chain is a collection of nodes such that all nodes in the collection are connected through either a verb or modifier node.
Reading a sentence chain

Figure 3.3 shows the order in which nodes and edges in a sentence chain should be read. The following steps should be taken when reading a sentence chain:

- **(0) Find the starting verb.** The starting verb is the first verb in the chain of verbs that make up a sentence chain. Starting verb nodes can be identified by locating a verb with no incoming edges from other verb nodes. In Figure 3.3, the starting verb is *may be,* and is annotated with a red "2".

- **(1) Find and read subjects of the starting verb.** Once we have determined the starting verb, we can begin to read the sentence. The node representing the subject of a verb points to the verb node. In this case, the *pull model* node, annotated "1" is the subject of the *may be* node. So, we read "pull model may be".

- **(2,3,4,5) Read all noun nodes before the next verb node.** Now, we read all the noun nodes that are attached to the *may be* node (2). We follow the left-most edge first to read *inefficient* (3). Then, we follow the next edge, and read its label *because* (4). Verb nodes are read only once in a sentence: We do not re-read the *may be* node. The *because* edge points to a modifier node, *classes.* The modifier node points to the node, *observer.* Modifier nodes are read together with the dictionary nodes they modify. In this case we read the phrase *observer classes* (5). So far, we have read “pull model may be inefficient because observer classes...”.

- **(6,7,8,9,10) Read the next verb node.** Once all noun nodes attached to a verb node are read, we read on to the next verb node, in this case *must ascertain* (6). We know that the *observer classes* node combination (5) is the subject of this verb node because of the blue arrow pointing into the verb node. There are no noun nodes attached to this verb node, so we read the label *what* on
the black edge to the next verb node (7), and then we read the verb node changed (8). The changed node has one outgoing edge, labeled from the (9). We read that edge, and then the node to which it points, the subject node (10). Finally, we have read “pull model may be inefficient because observer classes must ascertain what changed from the subject”.

3.1.3 Sequence Chains

To address the need to depict sequences of sentence chains in the graph, we augmented the pattern-level graph with sequence chains. A sequence chain links sentence chains that should be read in a particular order. These groups of sentence chains would typically include sequences of events, or a description of consequences of a previous statement. A sequence chain is represented by a chain of diamond-shaped nodes. Figure 3.4 shows an abstract representation of a sequence chain. This figure shows that sequence nodes are connected by thick, red edges, and each node in a sequence chain has an edge pointing to the first verb of a sentence chain. The first node in a sequence chain is labeled with the word “FIRST”, and all subsequent sequence chain nodes are labeled with the number of their position in the chain.²

3.2 Source-Level

The source-level in the DPRG depicts the structural relationships in the code base. Nodes in this level of the DPRG represent entities from the source, such as classes and methods. Edges represent relationships between entities, such as structural associations as in calls between methods, or inheritance relationships between classes. In the right most portion of Figure 1.3, three source-level nodes are shown, depicting that two of the nodes, PolyLineFigure and AttributeFigure, extend AbstractFigure,²

²The choice of the label “FIRST” is used, rather than “1”, so that sequence changes can be searched for using a unique string. More information about queries of this kind is given in Section 4.2.1.
Figure 3.3: Path of how to read a sentence chain. The green path shows the order in which edges are traced, the red numbers show the order in which nodes and edge labels are read. The sentence reads *a pull model may be inefficient because observer classes must ascertain what changed from the subject.*
and one, the standard package, contains the AbstractFigure class (depicted by the has relationship).

There are five source-level node types: package, interface, class, method, and field. There are six source-level edge types: has, calls, implements, extends, writes, accesses, and takes. An example of each relationship and node type is shown in Figure 3.5. At the source-level, packages are shown as double circles, interfaces as triangles, classes as diamonds, methods as rectangles, and fields as ovals. Edge types are further described in Table 3.1.

### 3.3 Link-Level

The link-level is depicted in the center portion of Figure 3.1, labeled Link-level. The link-level illustrates the relationships between entities from the design pattern and entities in the source. To depict this correspondence, the link-level contains nodes from the pattern-level, shown in the pattern structure area of Figure 3.1, nodes from the source-level, shown in the area overlapping the source-level, and edges linking
Table 3.1: Source-Level Edges

<table>
<thead>
<tr>
<th>Edge</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>has</td>
<td>Denotes containment. For instance, classes can have other classes, methods, or fields; methods can have fields; packages can have interfaces and classes.</td>
</tr>
<tr>
<td>calls</td>
<td>Control transfer. One method calls another method.</td>
</tr>
<tr>
<td>implements</td>
<td>Interfaces can be implemented by classes.</td>
</tr>
<tr>
<td>extends</td>
<td>Denotes subclassing. A class can extend another class.</td>
</tr>
<tr>
<td>writes</td>
<td>Indicates a change to a variable. A method can write to the value of a field.</td>
</tr>
<tr>
<td>accesses</td>
<td>Indicates a read of a variable. A method can access the value of a field.</td>
</tr>
<tr>
<td>takes</td>
<td>Indicates the type of a parameter. A method can take a class or field as a parameter.</td>
</tr>
</tbody>
</table>
Figure 3.5: Abstract Representation of the Source-Level
the two, shown in the *linking edges* area.

The left most portion of the graph shown in Figure 1.3 is an example of the link-level of a DPRG. In this figure, the pattern-level *subject* node (a filled box) has been associated with the AbstractFigure class, among others, in the code base.

The link-level of a DPRG has three components: a set of code-facts, which depict the code-level entities from the design pattern and their relationships to one another (pattern structure), a set of edges linking code-facts to nodes from the source-level (linking edges), and the source-level relationships between the linked source-level nodes.

### 3.3.1 Code-Facts

Pattern texts include descriptions of an abstract implementation of the design solution. This abstract implementation is described in structural diagrams in the pattern text, which depict relationships between structural entities such as classes and methods. Code-facts represent the relationships depicted in these diagrams.

A selection of the OBSERVER code-facts is depicted in Figure 3.6. The entities and relationships in this diagram come from the class diagram in the OBSERVER pattern depicted on page 294 of [23].

Code-fact entities can be classes, methods, or variables. These entities are represented as nodes in the link-level. The same relationships used in the source-level graph can occur between code-fact entities.

In the case that a code-fact entity is also a dictionary node in the pattern-level, it is represented at the link-level by the pattern-level node. Relationships between code-fact entities are the same as those between source-level nodes.

### 3.3.2 Links to Source-Level Nodes

The correspondence between pattern-level entities, as described by the code-facts, and entities in the source-level, is comprised of a collection of edges, the sources of
Figure 3.6: Portion of the OBSERVER Code-fact Graph
which are code-fact entities, and the destinations of which are source-level entities.

Code-fact entity nodes are linked to zero or more source-level nodes. Source-level nodes are linked to one or more code-fact nodes.

An example of these links is shown in Figure 3.7. On the left are shown the code-fact nodes, and on the right, the source-level nodes to which they are linked. For example, subject, a pattern-level node, is linked to two source-level nodes, the AbstractFigure and the FigureChangeEventMulticaster.

Figure 3.7: Links between a selection of Observer code-fact entities, and source-level entities.
3.3.3 Source-Level Nodes and Edges

The link-level contains source-level nodes that are linked to code-fact nodes. An edge from the source-level is included in the link-level if both the source and destination nodes of that edge are included in the link-level. Figure 3.8 shows a portion of the link-level for the OBSERVER pattern. In this figure, takes edges are visible between the FigureChangeListener and the remove and add methods which take instances of that class as an argument. These edges are visible because the source and destination nodes are both in the link-level.

3.4 Summary

In this chapter we have described the DPRG model. The DPRG consists of three levels: the pattern-level, the source-level, and the link-level.

The pattern-level is a graphical representation of the text of a design pattern. It consists of two structures: sentence chains, and sequence chains. Nodes in the pattern-level represent phrases from the design pattern text. Phrases related to keywords are represented by single nodes in the pattern-level graph. Other phrases are each represented by separate nodes.

The source-level is a graphical representation of the entities and relationships between entities in a code base. Types of entities include classes, methods, and fields. Types of relationships include calls, has, and writes.

The link-level overlaps the pattern-level and source-level, and depicts links that indicate that nodes from the pattern-level correspond to nodes at the source-level.
Figure 3.8: Portion of the Link-level for the OBSERVER Pattern
Chapter 4

Design Pattern Rationale Graph Tool

This chapter describes how a developer interacts with the DPRG tool and describes the implementation of the DPRG tool. As a basis for description, we refer to the example from Section 1.3.

4.1 Preparation of Input Files

Before a developer can use the DPRG tool, several files must be present. This section describes the files that are necessary for the creation of the three levels of the DPRG.

4.1.1 Pattern-Level Files

The creation of the pattern-level of a DPRG requires two inputs: the text comprising the pattern, and a dictionary of design elements—participants and concepts—specific to the pattern.

First, the developer provides the pattern text. The text can be extracted from a digital representation of the pattern. The developer must annotate the text
to include sequential information by adding the word, "first", to the beginning of the first sentence in a set of steps, and the word, "then", to the beginning of each subsequent sentence. We have not found this step to be onerous; it took less than 10 minutes to annotate the text of the OBSERVER pattern.

Next, the developer provides a dictionary of keywords. To help build this dictionary, the DPRG tool can extract a list of noun phrases found in the pattern text. The developer then peruses this list, and identifies those noun phrases that are design elements. As described in Section 3.1, the choice of design elements dictates the structure of the DPRG: A noun phrase identified as a design element corresponds to one node in the DPRG; instances of other noun phrases are represented by separate nodes. For the OBSERVER DPRG, the design elements identified included change request, notify and observer. Nouns not chosen as design elements included need, state, class, and call.

The pattern-level files need to be set up only once per pattern. Once prepared, they can be linked to multiple source-level graphs. Since the choice of dictionary elements affects the structure of the pattern-level graph, a developer using the DPRG tool may wish to add or delete design elements to or from the dictionary when using the tool. Regenerating the DPRG for the OBSERVER pattern takes under 30 seconds on a 600MHz Intel Pentium-3 processor system, running Windows 2000.

4.1.2 Source-Level Files

The source-level of a DPRG is dependent upon the system of interest, not the pattern. To create this level, a developer runs a program-database tool which extracts information about entities and relationships from a code base, such as the calls relationship between methods, and the inheritance relationship between classes, amongst others. These entities and relationships form the source-level of the DPRG.

\footnote{The time required for this activity is reported in Section 5.3, but is generally within 15 minutes regardless of the size of the pattern.}
The DPRG tool relies on the use of third-party program databases for this information. Currently, the DPRG tool is able to post-process output from the Chava [40] tool, which provides program database information for Java [2] code. In the case of the JHotDraw code base (described in detail in Section 5.3.1) this step takes about five minutes on a system with a 1GHz Intel Pentium-3 processor, running Redhat Linux 7.3.

4.1.3 Link-Level Files

In order to form the link-level of a DPRG, the DPRG tool requires a code-fact file. This file describes the relationships between selected pattern-level nodes mentioned in the structure portion of the design pattern. Code facts are described in detail in Section 3.3.1. Since these graphs are small, and the number of participants involved is also small, these graphs generally take under 15 minutes to translate from the original design pattern text. However, our DPRG tool could be extended to automatically analyze and extract this information from a digital representation of a pattern.

Figure 4.2 shows the code-fact file for the OBSERVER pattern. The contents of this file came from two diagrams in the OBSERVER pattern: the class relationship diagram [23, page 294], and the collaboration sequence diagram [23, page 295]. Figure 4.1 is an example of such a diagram. It displays the relationships between the classes described in the pattern.

4.2 Interaction with the DPRG Tool

A developer performs two kinds of activities when interacting with the DPRG tool (Figure 4.3). First, a developer performs queries, using the query view, to search and browse through the three levels of the DPRG. When interacting with this view developers can manipulate the view of the DPRG by adding to the view, subtracting from the view, and intersecting views. These actions are depicted as +, −, and ∩,
Figure 4.1: Class Diagram for the OBSERVER Pattern
Figure 4.2: Code-fact File for the OBSERVER Pattern
in Figure 4.3. To begin the query process, the developer provides a DPRG.

The second activity is the creation of the link-level of the DPRG using the linking process. A developer begins by suggesting an initial link, or seed, to relate the pattern-level to the source-level of a DPRG. During this process, developers accept and reject links, and ask the tool for more links for the developers to assess. The querying and linking actions can possibly be interleaved. We first describe the query actions available, and then describe the linking actions. The tools the developer uses for these activities, called the query view and the link view, are provided as extensions to the GraphViz [41] program.

4.2.1 Performing Queries on a DPRG

Performing a query results in a query view of the DPRG. Figure 4.4 illustrates the query process. The developer begins at the step labeled START by explicitly specifying the files for each level of the DPRG.

The initial view is blank. From this view, the developer can perform a regular expression query or can load a previously saved view. Once the view is populated, the developer may perform the node-specific functions of expansion and subtraction and intersect the view with a previously saved view.

We now describe each query view mechanism in more detail.
Figure 4.4: User interaction model for performing queries on a DPRG. The user provides the three levels of the DPRG to the tool, and a is shown blank view. Chains can be added to the view by loading an existing view, or performing a regular expression query. Once the view is populated, chains can be subtracted from the view, or added to the view (using regular expression or node expansion queries). Views can also be intersected and saved.
A regular expression query applies to the pattern-level. Such a query results in the inclusion of all sentence chains containing nodes that contain the expression. Also included are sequence nodes pointing into those sentences. Figure 4.5 shows partial results from a regular expression query for *efficien* from the example in Section 1.3. Two sentence chains are revealed. The first sentence, on the left, reads “You can improve update efficiency by extending the registration interface to allow for registering observers, but only for specific events of interest”. The second sentence, on the right, reads “The pull model may be inefficient because observer classes must ascertain what changed from the subject”.

Figure 4.5: DPRG view of regular expression query for *efficien*
Node Expansion

Once the query view is populated, a user can choose to expand a node. Expanding a node adds the context of that node to the query view. The nature of this context depends on the way in which the node was expanded.

A developer can choose to expand a node in any of the three levels: pattern, source or link. A node can only be expanded at a level in which it is contained. Some nodes may only be expandable at one level such as the pattern or source. Some nodes may be expandable at two levels, either the pattern and link, or source and link.

At the pattern-level, sequence and dictionary nodes can be expanded. When a node is expanded, the context added consists of all sentence chains containing that node. In the case of a sequence node, all of the sentences in the sequence will be added to the view. Since only sequence nodes and dictionary noun nodes may be contained in more than one chain, expanding any other DPRG node would not result in the addition of any new chains.

At the link-level, two kinds of nodes can be expanded: pattern-level nodes specified in the code-fact file, and source-level nodes linked to the pattern-level nodes. When a node is expanded at the link-level, the context added includes all of the nodes that share an edge with the expanded node. Edges between neighbouring nodes are also added. Figure 4.6 shows the results of the link-level expansion of the subject node from the graph shown in Figure 4.5.

Source-level node expansion allows a developer to browse the source-level, viewing the relationships between classes, methods and fields. When a node is expanded at this level, the context added consists of all of the nodes that share an edge with the expanded node, and the edges between those neighbouring nodes.
Sentence Subtraction

To support browsing at the pattern-level, sentence chains can be deleted from a view. When a node is subtracted, edges in the sentence chain or chains in which the selected node occurs are deleted from the view. However, the nodes in the chains are not deleted from the view: They are orphaned, and are moved to the side of the view to allow re-expansion of those nodes during subsequent investigation.

View Intersection

To help narrow pattern-level expansions and searches, two pattern-level graphs can be intersected by the tool. If a developer wishes to see all of the sentences containing two or more regular expressions, the developer performs each query in a separate view, and then intersects the resultant views.
Figure 4.7: User interaction model for forming the link-level of a DPRG. A developer can ask the tool to infer link possibilities, and can accept, reject and view link possibilities suggested by the tool. The developer can also save the set of possibilities, thus forming the link-level for a DPRG.

Load and Save: Revisiting Prior Queries

A developer can save a view to be subsequently re-loaded or to be used in an intersection operation.

4.2.2 Linking the Pattern and Source Model Levels

Links between the pattern- and source-levels are established through an iterative and incremental process. During this process, a developer interacts with the link view of the DPRG tool. Figure 4.7 shows that a developer suggests one or more initial links, or seeds, from the pattern-level to the source-level (depicted as infer(1)). The DPRG tool can then suggest additional links (depicted as link possibilities) each of which may be viewed, accepted or rejected by a developer. This process is repeated until a developer is satisfied that enough of the design pattern elements have been mapped to the code base of interest (infer(2)). The algorithm used for inferring
Figure 4.8: Initial seed input to the DPRG tool for linking JHotDraw to the OBSERVER pattern.

matches is provided in Section 4.3.3.²

Seed Format

To provide the initial seed or seeds for the link inference process, a developer must specify both a pattern-level element and a source-level element.

For instance, a developer wishing to link the JHotDraw code base to the OBSERVER pattern might suggest a link from the observer class to a class they posit may play the role of an observer in JHotDraw. Two such seeds are shown in Figure 4.8. In the seed, the developer must provide enough information about the source-level node for the DPRG tool to identify it. At the source-level, all elements in the program database are assigned IDs. One way to identify a node is to explicitly specify this ID. A developer can find a node ID by searching through the source-level file itself. This approach is shown in the first line of Figure 4.8. A developer can also specify the name and node type of the target source-level node. The second line of Figure 4.8 shows this approach; the interface called FigureChangeListener is associated with an observer. The DPRG tool uses the name and node type to identify the ID.

²The time required to link a particular source-level to a pattern-level is discussed in detail in Chapter 5. Since this activity is generally intertwined with querying, it is difficult to present specific estimates of time required. Times may range from 10 minutes, for a developer who is somewhat familiar with a code base, to over an hour for a developer who is using the activity to explore the code base.
Inferred Links

Having provided an initial seed or set of seeds, a developer can use the DPRG tool to infer additional links between the pattern- and source-levels. This action is depicted as infer(1) in Figure 4.7. After a developer has suggested the seed of observer to FigureChangeListener, the DPRG tool infers several link possibilities:

\[ N, 0.66, "attach", "addFigureChangeListener" \]
\[ N, 0.4, "notify", "invalidate" \]
\[ U, "subject" \]
\[ U, "update" \]
\[ U, "subject state" \]

Like a seed, a link is a pair, consisting of a pattern-level node and a source-level node. Links also provide additional information to the developer. Any new link inferred is flagged with an “N” in the link view, and is shown with a correspondence rating which indicates how well the graph neighbouring the source-level node of the link corresponds to the graph neighbouring the pattern-level node of the link. This rating is discussed further in Section 4.3.4.

Pattern-level nodes that are annotated with a “U” indicate code-fact nodes not yet linked to the source-level. As shown in Figure 4.7, a developer can accept and exclude inferred links. Links are annotated with an “A” when accepted, and an “X” when excluded. Excluded links are not considered if inference is requested. At any time, a developer can choose to re-accept an excluded link.

The action labeled infer(2) in Figure 4.7 shows a request for additional link possibilities based on the accepted inferred links. The following list of links was inferred after the attach and notify method suggestions above were accepted by the developer. In this case, the new link possibility of the subject to the Figure class was inferred.

\[ N, 0.75, "subject", "Figure" \]
A developer continues seeding, inferring, and excluding links until they feel that the pattern is satisfactorily linked to their source-level. Since the linking process is typically incremental, and is interleaved with pattern-level queries, the developer may choose to stop once relevant portions of the pattern-level are linked to source. For example, the developer may choose to stop the linking process when all pattern-level nodes are linked, or when no new links are inferred by the tool.

**Link Inspection**

From the link view, a developer can inspect a link to provide more information on whether a link should be accepted or excluded. Inspecting a link results in a view of a portion of the DPRG. Figure 4.9 is an example of such a view. The view shows the pair of nodes comprising the link, and shows the nodes neighbouring the pattern-level node and the source-level node in the link. In this case the link being viewed is between the `attach` method and the `addFigureChangeListener` method from the link-level.

The nodes and edges neighbouring the source-level node come from the source-level of the DPRG, whereas the nodes neighbouring the pattern-level node are drawn from the code-fact file. Also included are all edges between neighbouring nodes. For instance, the source-level portion shows all of the nodes with relationships to the `addFigureChangeListener` method. Also shown are relationships between the neighbouring nodes. This contextual information helps the developer determine if the link seems appropriate.

The correspondence rating of the link in Figure 4.9 is shown on the bold edge connecting the two portions of the graph. The rating of "1" indicates that
Figure 4.9: Inspection of the link attach method to addFigureChangeListener

all of the edges related to the pattern-level node attach have corresponding edges to the source-level node addFigureChangeListener. The pattern-level portion of Figure 4.9 shows that the attach method belongs to the subject class, and that it requires the observer as a parameter. The source-level portion shows that the addFigureChangeListener belongs to a class called AbstractFigure, and takes an interface, the FigureChangeListener, as a parameter.

When inspecting a link, a developer can perform any of the query process actions.

4.3 DPRG Computation

This section describes the implementation details of the DPRG tool. First, we describe the formation of the pattern-level of a DPRG. Next, we explain the com-
putation of a node expansion. Finally, we describe the two main computations in
the linking process, inference and correspondence rating.

4.3.1 Forming the Pattern-Level Graph

Given the annotated pattern text and the dictionary, the DPRG tool automatically
creates the pattern-level of the DPRG.

The DPRG tool takes the pattern text and uses a parts-of-speech tagger,
called LTCHUNK [47], to identify the noun and verb phrases in each sentence of
the pattern text. Each sentence is processed individually. Except for noun phrases
that appear in the dictionary, each occurrence of a noun or verb phrase introduces a
new node into the pattern-level of the DPRG. The first node identified in a sentence,
whether a noun or a verb phrase, is considered a source node. Each subsequent node
encountered in the same sentence is considered a destination node, and introduces
an edge between the source and destination node. When a node based on a verb
phrase is encountered, the source node is reset to the verb phrase node. Edges are
labeled by any phrase linking the noun and verb phrases.

As an example, consider the following sentence from the OBSERVER pattern,
which has been processed by LTCHUNK, and which shows noun phrases in double
square brackets and verb phrases in double parentheses:

[[ all observers ]] are (( notified )) whenever the [[ subject ]] (( under-
goess )) [[ a change ]] in [[ state ]] [23, Paragraph 2, page 294].

The fragment of the DPRG created from this sentence is shown in Figure 4.10.
Nodes and edges are assigned colours and connections based on their description in
Section 3.1.

4.3.2 Computing a Node Expansion

In Section 4.2 we explained that node expansion and regular expression searches
both result in query views. Now we will describe how a query view is generated.
There are several instances during the query process in which a developer asks to add or subtract sentence chains from the view. At times, the developer may wish to add one or more sentence chains that contain a regular expression or contain a particular node.

As was described in Section 3.1, sentence chains consist of a chain of verb and modifier nodes. Nouns nodes and sequence markers are attached to the chain. Within the context of one sentence chain, verb and modifier nodes are considered non-terminal, while nouns and sequence markers are considered terminal. Verb nodes and modifier nodes are non-terminal because they form the structure of the chain. Noun nodes and sequence markers are attached to the chain as leaves.

Figure 4.3.2 describes the recursive algorithm to alter a DPRG view to include all chains starting from a particular node. The algorithm begins by initializing the set of nodes to expand (1.a). If the developer specifies a regular expression query,
**SET** is initialized to contain all nodes containing the regular expression. Otherwise, **SET** contains the node selected for expansion. Next, **VIEW** is initialized to the nodes currently visible in the DPRG view (1.b).

The algorithm continues by expanding **SET** to include any nodes neighbouring nodes in **SET**, with the exception of dictionary and sequence nodes (2.b). This process continues until **SET** reaches a fixpoint (2.a). Dictionary and sequence nodes are not considered because they may span more chains than intended. If these nodes were considered, more than one sentence would be added to **SET**.

Once a fixpoint is reached, dictionary and sequence nodes that are connected by an edge to any of the nodes in **SET** are added (3). In the special case where a sequence node is selected for expansion, the stipulation that sequence nodes cannot be added during step 2.b is ignored, so that only dictionary nodes are considered terminal.

This algorithm does not compute single sentence chains. The result of this algorithm will be all of the sentence chains containing the node(s) in the initial **SET**. If the initial **SET** was comprised of one node that appeared in only one sentence chain, then the result of this algorithm would be one sentence chain. However, if the initial **SET** included multiple nodes, or a node that appears in multiple sentence chains, then the result would be a collection of sentence chains.

### 4.3.3 Link Inference

The DPRG tool infers additional links based on links that have been seeded or accepted by a developer. Table 4.1 summarizes the definitions used in the explanation of the inference and rating of links. The pattern- and source-level graphs are defined as *P* and *S*, respectively. *link* is a pair, (*p*, *s*) that links a pattern-level node to a source-level node. *link*(*p₀*, *s₀*) is a pair that has either been seeded or accepted by a developer. *link*(*p₀*, *s₀*) is the input to the inference algorithm.

*edge*(*a*, *b*) is an edge between two nodes. The label and direction of that edge
1. Initialize:  
   1.a SET = node(s) to expand  
   1.b VIEW = current nodes in the DPRG View  
2. a Repeat until: SET stops changing  
   {  
   2.b all nodes (except dictionary and sequence nodes)  
   sharing an edge with a node in SET are added to SET  
   }  
3. all dictionary and sequence nodes sharing an edge  
   with any node in SET are added to SET  
4. SET is added to VIEW  
5. all edges where both source and dest are contained in SET are  
   added to VIEW  

Figure 4.11: Algorithm to Compute a Node Expansion

are defined by label and direction, respectively. same is a test for the equivalence of  
two edges, and is true if the two edges have the same label and direction.

$I_O$ is the set of links inferred by the DPRG tool based on $\text{link}(p_0, s_0)$, such  
that an inferred pattern-level node, $p_1$, has an edge to or from $p_0$, and an inferred  
source-level node, $s_1$, has an edge to or from $s_0$. The direction and label of the edge  
between the pattern-level nodes must match the direction and label of the edge  
between the source-level nodes.

$$I_O = \{ \text{link}(p_1, s_1) \mid \text{same}(\text{edge}(p_1, p_0), \text{edge}(s_1, s_0)) \land (p_1 \in P) \land (s_1 \in S)\}$$

The formation of the set $I_O$ is depicted in Figure 4.12. The two white areas  
represent nodes in the sets $P$ and $S$. The edges in the grayed area are in $I_O$. A  
link is in $I_O$ when a node in $S$ has the same relationship to $s_O$ as the node in $P$  
has to $p_O$. For instance, pattern-level node 2 has an edge labeled calls pointing into  
$p_O$, and both source-level nodes A and B have edges labeled calls pointing into $s_O$.  
Hence, two link possibilities are inferred: 2—A, and 2—B.

Pattern-level nodes are not typed, meaning that they are not declared as  
classes, methods, fields, or any other structural entity. Based on this ambiguity, any  
pattern-level node can be linked to any source-level node.
Figure 4.12: Edges between the pattern-level graph $P$ and the source-level graph $S$ show inferred links. The set of bold links is $\mathcal{I}_O$, and indicate inferred links between pattern-level and source-level nodes. The dotted line indicates the original link either suggested as a seed or previously accepted by a developer.
Table 4.1: Definitions Used in Inference and Correspondence Rating

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>set of nodes in a source-level graph</td>
</tr>
<tr>
<td>( P )</td>
<td>set of nodes in a code-fact graph</td>
</tr>
<tr>
<td>( \text{link}(p, s) )</td>
<td>link from ( p ) to ( s ) such that ( p \in P ) and ( s \in S ).</td>
</tr>
<tr>
<td>( \text{link}(p_0, s_0) )</td>
<td>seeded or accepted link</td>
</tr>
<tr>
<td>( \text{link}(p_e, s_e) )</td>
<td>link to be rated</td>
</tr>
<tr>
<td>( \text{edge}(a, b) )</td>
<td>edge between ( a ) and ( b )</td>
</tr>
<tr>
<td>( \text{direction}(\text{edge}(a, b)) )</td>
<td>direction of ( \text{edge}(a, b) )</td>
</tr>
<tr>
<td>( \text{label}(\text{edge}(a, b)) )</td>
<td>label of ( \text{edge}(a, b) )</td>
</tr>
</tbody>
</table>
| \( \text{same}(\text{edge}(a, b), \text{edge}(c, d)) \) | \((\text{direction}(\text{edge}(a, b)) \equiv \text{direction}(\text{edge}(c, d))) \land \)
|                           | \((\text{label}(\text{edge}(a, b)) \equiv \text{label}(\text{edge}(c, d)))\)          |

4.3.4 Determining the Correspondence Rating of a Link

Once links have been inferred, they are given a rating indicating how well the graph neighbouring the source-level node of the link matches the graph neighbouring the pattern-level node of the link. To calculate this rating, we examine the nodes neighbouring the link to be evaluated. We then report the percentage of edges between the neighbouring nodes in the pattern-level that match edges between the neighbouring nodes at the source-level.

Let \( \text{link}(p_e, s_e) \) be the link to be evaluated. The evaluation begins by generating a set of mappings, \( M \), from nodes neighbouring \( p_e \) to nodes neighbouring \( s_e \). This set is defined in the same way as \( I_0 \):

\[
M = \{ \text{link}(p_1, s_1) \mid \text{same}(\text{edge}(p_1, p_e), \text{edge}(s_1, s_e)) \land (p_1 \in P) \land (s_1 \in S) \}
\]

Patterns are often implemented using more inheritance than was described in the original pattern structure. For example, in the OBSERVER structure, only one level of inheritance is used, in which the concrete observer subclasses the observer class. In practice, however, there may also be subclasses of the concrete observer. In this case, both levels of concrete observer would subclass the observer class.
(A) Links are included in M+

(B) A subset of links (shown as solid) from M+ make up an overlay, O

(C) Edges between neighboring nodes are compared between Eo and Ep

The number of edges in Ep also in Eo determines the rating

Figure 4.13: Sets Used in Calculating a Correspondence Rating for a Link \((p_e, s_e)\)
subclass of concrete observer, would not have a direct extends edge from it to the observer class. To take this into account when assigning a correspondence rating, we generate a set, $M_S$, of additional possible substitutions. If $s_e$ of the link to be evaluated is a method and overrides $s_e \in S$, or $s_e \in S$ is a class that subclasses $s_e$, then we consider all of the nodes neighbouring $s_e$ to be possible substitutions for a node neighbouring $p_e$.

$$M_S = \{ \text{link}(p_2, s_2) \mid \text{same}(\text{edge}(p_2, p_e), \text{edge}(s_2, s_e)) \land (p_2 \in P) \land (s_2 \in S) \}$$

The sets $M$ and $M_S$ are both considered when calculating the correspondence rating. These two sets are combined into a set $M_+$:

$$M_+ = M \cup M_S$$

This set is abstractly depicted in Figure 4.13, portion (A) in the grayed area labeled $M_+$. The powerset of $M_+$ ($\mathcal{P}(M_+)$) is a set of all possible combination of links in $M_+$. Next, we determine all overlays of the pattern-level onto the source-level. An overlay, $O$, is a set of links such that no pattern-level or source-level node appears more than once. The formation of the set $O$ is depicted in Figure 4.13 (B).

$$O \in \mathcal{P}(M_+) \land \exists((\text{link}(p_i, s_j) \in O) \land (\text{link}(p_k, s_l) \in O) \land (p_i = p_k) \land (s_j = s_l))$$

Each overlay is evaluated in terms of how many of the edges between nodes mapped from the pattern-level are present between nodes mapped at the source-level.

A pattern-level edge between nodes linked in $O$, is present between mapped source-level nodes linked in $O$ if the relation similar holds. similar indicates that for a pattern-level edge, $\text{edge}(p_a, p_b)$, to be similar to a source-level $\text{edge}(s_a, s_b)$, $p_a$ must be linked to $s_a$, $p_b$ must be linked to $s_b$, and the edges must share the same label and direction.

$$\text{similar}[\text{edge}(p_a, p_b), \text{edge}(s_a, s_b)] = [\text{link}(p_a, s_a) \in O, \text{link}(p_b, s_b) \in O]$$
$$\land [\text{same}(\text{edge}(p_a, p_b), \text{edge}(s_a, s_b)))]$$
The correspondence rating of an overlay $O$ is then computed by dividing the number of similar edges, $E_O$, by the number of edges between nodes neighbouring $p_e$, $E_p$. In Figure 4.13 portion (C), $E_p$ has two edges, one labeled calls and one labeled has. $E_O$ has one edge labeled calls. Two dashed edges are shown in $E_O$. These represent edges between the source-level nodes linked in $O$ that were not found to be similar to any edge in $E_p$. One source-level edge is labeled calls, and is similar to the pattern-level edge labeled calls. There is no edge similar to the edge labeled has in $E_p$.

The set $E_O$ is defined as:

$$E_O = \{\text{edge}(p_i, p_j) \mid \text{similar}(\text{edge}(p_i, p_j), \text{edge}(s_i, s_j))\}$$

The set $E_p$ is defined as:

$$E_p = \{\text{edge}(p_i, p_j) \mid \exists \text{link}(p_i, s_k) \in O \land \exists \text{link}(p_j, s_l) \in O\}$$

An overlay rating is thus:

$$\frac{|E_O|}{|E_p|}$$

Since $|E_p|$ will always be greater than or equal to $|E_O|$, the overlay rating will never be greater than 1. In Figure 4.13, the rating for link $p_e-s_e$ is shown as .5, since only one of the two edges in $E_p$ was also in $E_O$.

The correspondence rating for link $(p_e, s_e)$ is determined by taking the maximum overlay rating of all possible overlays.

### 4.3.5 Performance

The computation time for the pattern-level graph creation increases with the size of the pattern. The longest time for a graph production was 30 seconds on a system with a 600MHz Intel Pentium-3 processor, running Windows 2000.

The potential exists that the link inference and rating computation algorithms cannot be computed in polynomial time. Formalizing the worst case for
these computations is impractical, since they depend on both the number of nodes neighbouring the pattern- and source-level nodes, and the number of possible substitutions between the graphs. Worst case, or even typical numbers, for these variables are not known. In practice, pattern-level graphs are bounded since the number of participants in a pattern is small. At times, a source-level graph may be large. Still, only subgraphs of source-level graphs are considered in link inference and rating computations. The number of nodes and edges in a subgraph is determined by the number of nodes neighbouring a linked node. One of the largest examples we have worked on, the Zen code base, contained a subgraph of 100 nodes and edges. Even in this case, the inference and rating computations each took under a minute to complete on a machine with a 1.6GHz Intel Pentium-4 processor running Windows 2000.

4.4 Summary

In this chapter we have described the DPRG tool.

To apply the DPRG tool, a developer begins by preparing input files. This includes massaging the text of the design pattern to include sequence information, forming a source-level graph of a code base, and specifying code-facts from the structural portions of design patterns.

A developer engages in two main activities when interacting with the DPRG tool. First, a developer uses the DPRG tool to link the pattern-level to the source-level. Second, a developer performs queries such as node expansion and regular expression searching to form views of a DPRG with which to explore design goals.

We also described three implementation level details: the approach for formation of the pattern-level graph, the algorithm for suggesting link possibilities to a developer, and the formalization of the correspondence rating of the links.

Finally, we report on the performance of the DPRG tool. In the examples we have tried, formation of the pattern-level graph is computed within seconds.
Although the inference and rating of links has the potential to be computationally intensive, the performance of these algorithms is acceptable due to the nature of the graphs upon which they operate.
The thesis of this work makes two main claims:

- **Completeness:** a developer using the DPRG tool will be able to identify design goals relevant to their code more completely.

- **Confidence:** a developer using the DPRG tool will have a higher degree of confidence with relation to how those goals are carried out in a code base.

The thesis makes two additional claims:

- **Lightweightness:** the DPRG tool is lightweight; the time required to apply the approach is constrained within the time required for a task.

- **Efficacy:** the pattern-level representation of the design pattern allows developers to navigate through a design pattern and to extract design context and detail pertaining to pattern entities and concepts.

To validate the main claims of the thesis we conducted two case studies. The first was designed to measure the degree to which a developer is able to completely report relevant design goals for a pattern implementation. The second was designed to evaluate developers' confidence about the implementation of design goals after using the DPRG tool.
We chose to validate these claims with case studies to provide a high degree of realism. The case study code bases needed to be sufficiently large to guarantee the presence of interacting and crosscutting design goals, and the systems had to be implemented with design patterns. The case study format chosen allowed us to carefully observe all interactions with the tool, and also all knowledge gained through its use, and to assess the evidence supported by that data with relation to the initial hypotheses. This format was chosen over an experiment because it is not possible to find a set of participants who are sufficiently familiar with the same code base and the same task.

To evaluate completeness, two developers from industry were involved in determining the goals related to a large code base: the Zen CORBA ORB [76]. The developers were both experts with the Zen code base. We then compared the design goals identified by these developers to those specified by a (non-expert) developer using the DPRG tool.

To evaluate confidence, we constructed a study with two cases, each involving a developer from industry. The subjects were chosen because they could report on their experience using the DPRG tool to revisit a task they had previously completed as part of their work on an industrial code base. Based on their reports, we evaluated whether they obtained information from the tool that would affect their confidence with relation to the design goals of their systems.

To validate the additional claims of lightweightness and efficacy, we undertook two small studies: a study with two cases to evaluate the lightweightness of the tool, and an experiment to compare how well developers reading a pattern-level graph were able to report design detail and context of a design pattern in comparison to developers reading the design pattern itself. The lightweightness study examined two case subjects from industry. Each subject was asked to create a DPRG and use it for an investigation task. The comparative experiment on pattern-level efficacy involved eight subjects: four from industry, and four from academia. Subjects
were asked to respond to questions about design patterns. Half of the subjects were given the DPRG tool and half were given the design pattern. Their responses were compared in terms of levels of detail and context with relation to the entities and concepts in question.

We used a range of patterns in these studies. These are described in Table 5.1.

In this chapter, we first describe the validation of the main claims of the thesis: a study to evaluate whether developers are able to use the DPRG tool to identify relevant design goals completely, and another study to evaluate developers' confidence with relation to design goals after using the DPRG tool. Then we report on the validation of the additional claims: a study to evaluate lightweightness of the DPRG tool, and a comparative experiment on pattern-level efficacy.

5.1 Case Study: Completeness

To verify that relevant design goals could be identified from portions of source code, we conducted a small study involving two experts and one DPRG tool user. In this study, the experts worked together to report design goals relevant to certain portions of the Zen CORBA ORB code base: a system in which design patterns have been heavily applied. The goals reported were specific to the code base, and were not necessarily dictated by a design pattern, although the portions of the code base used were implementations of particular design patterns. The tool user used pre-made DPRGs, and reported the design goals found through “upwards” expansion queries: from the link-level to the pattern-level. The experts compiled their list of goals from their knowledge of the code and of the design patterns. They did not employ the use of the DPRG tool to assist them. The investigator then compared the goals generated by the experts with those generated by the tool user.

We chose three design patterns from the Zen code base: CACHING [37], THREAD-SPECIFIC STORAGE [11] and STRATEGY [23]. The tool user investigated each pattern separately.
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
<th># Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORWARDER RECIPIENT [11]</td>
<td>Supports transparent interprocess communication using a peer-to-peer interaction model.</td>
<td>16</td>
</tr>
<tr>
<td>MONITOR OBJECT [61]</td>
<td>Synchronizes concurrent method execution within an object, ensuring only one method of the object runs at any given time.</td>
<td>26</td>
</tr>
<tr>
<td>REACTOR [61]</td>
<td>allows event-driven applications to handle service requests sent to applications by one or more clients</td>
<td>22</td>
</tr>
<tr>
<td>VISITOR [23]</td>
<td>supports the selection of a method to execute based on both the type of the initial recipient of a message and the type of the sender of that message—the caller</td>
<td>14</td>
</tr>
<tr>
<td>CACHING [37]</td>
<td>Reduces resource acquisition by storing the resource's identities for future use</td>
<td>9</td>
</tr>
<tr>
<td>THREAD SPECIFIC STORAGE [11]</td>
<td>Saves locking overhead and allows multiple threads to retrieve thread-specific data from one global access point</td>
<td>22</td>
</tr>
<tr>
<td>OBSERVER [23]</td>
<td>Enables all dependents upon an object to be notified when that object changes state</td>
<td>12</td>
</tr>
</tbody>
</table>
5.1.1 Design

This section describes the design of the study. We describe the subjects of the study, the code base used, the patterns used as a basis for the study, the format of the study and the data collected.

Subjects

This study involved two experts with the Zen CORBA ORB code base, and who were developers from Siemens AG, and one University of British Columbia faculty member using the DPRG tool for the first time.

Zen CORBA ORB Code Base

The Zen system is a CORBA Object Request Broker (ORB) written in Java. It is comprised of approximately 50,000 lines of commented code spread over 450 classes. Zen is the Java "child" of ACE/TAO, a CORBA ORB written in C++. ACE/TAO was the first main application for most of the patterns described in the Pattern-Oriented Software Architecture texts [11, 61]. As such, Zen employs many of the same patterns, and is also extended to work with real-time CORBA and real-time Java.

Patterns Chosen

Three patterns were chosen for this investigation: CACHING [37], STRATEGY [23], and THREAD SPECIFIC STORAGE [11]. We focused on these patterns because they have different authors, and because they range in length.

The STRATEGY pattern can be used to define a family of similar algorithms and to make those algorithms interchangeable through encapsulation. This pattern allows algorithms to change without affecting the clients that use the functionality. STRATEGY provides several useful features, including a way to configure the behaviour of a class, and a way to define variants of an algorithm.
STRATEGY has three main participants: the strategy, which declares an interface shared by all supported algorithms; the concrete strategy, which implements a specific algorithm using the strategy interface; and the context, which is configured with a concrete strategy object and which holds a reference to the main strategy object. The context may also define an interface to provide access to the main strategy.

The STRATEGY pattern covers ten pages of text, and is drawn from the text Design Patterns: Elements of Reusable Object-Oriented Software [23].

The THREAD-SPECIFIC STORAGE design pattern provides a design solution in which multiple threads can use one access point to retrieve thread-specific data. This global access point allows the data to be retrieved without locking overhead. There are two main issues that drive this solution: potential infeasibility of changing interfaces to legacy systems in which a single thread of control is assumed; and the need for a globally visible access point that is logically shared by multiple threads, though the data itself is thread specific.

The THREAD SPECIFIC STORAGE pattern has four participants: the thread specific object which is only accessible from within a particular thread; a collection, which is a set of all thread specific objects; a proxy, which provides an interface for creation and retrieval of a thread specific object; and application threads, which use the proxies to access the thread specific objects.

THREAD SPECIFIC STORAGE covers 22 pages of text, and is written in the Pattern-Oriented Software Architecture style [11].

The CACHING pattern describes a solution for avoidance of re-acquisition of resources. Re-acquisition is expensive, and is caused when resources are not released immediately after their use. In this solution, resources are kept in a cache, and are re-acquired from this cache to avoid re-acquisition. The four main goals the CACHING pattern addresses are: performance improvement; avoidance of cache usage complexity, avoidance of implementation complexity; and durability in terms
of changes to the source code.

CACHING has four main elements: the resource, which is an entity such as memory or a connection; the resource environment, which manages several resources; the resource cache, which buffers and evicts resources; and the resource user, which acquires the resource from the resource environment.

The CACHING pattern is written in the Pattern Oriented Software Architecture style [11] and covers nine pages of text.

Format and Data Collected

The study was conducted in two phases: the design goal collection phase, and the design goal discovery phase. The collection phase was performed by the two domain experts. The experts worked together to locate portions of the Zen code base that implemented design patterns. The experts then noted design goals that pertained to these portions of code. It was not stipulated that the design goals be contained in the relevant design patterns. The experts were asked to think of reasons the code was implemented the way it was.

The following task was given to the experts:

• Take a little time and think of all the design goals that are relevant for the code related to each pattern.

Design goals can be big, such as “distribution”, or smaller, such as “blocking”. They don’t necessarily align with structural things; for instance, in the OBSERVER pattern, the “observer” would not be considered a design goal, while “efficiency” would.

• Please come up with a list of the goals, and also short explanations of why these goals seem relevant for this piece of code.

The experts then compiled their list of goals based on their knowledge of the code base and the design patterns. They did not employ the DPRG tool to assist
them in this task.

The DPRG tool user was then given complete DPRGs for the three patterns, and was asked to perform upward expansions from source-level nodes in the link-level, and to note any design goals discovered. The DPRG tool user was also asked to note the sentences in which a goal was found.

We then compared the results of the DPRG tool user with the design goals explicitly specified by the experts. After this comparison, we asked the experts to review the list of goals found by the user, and to either accept or reject them.

We collected both explicit and implied goals from the DPRG tool user. The DPRG tool user was asked to report the design goals and the supporting sentence chain. Goals reported by the user were large, such as "efficiency" or "maintainability". These large goals sometimes encompassed smaller goals that were also specified in the chain. For instance, the user may have summarized the hypothetical sentence "efficiency with respect to data retrieval", as pertaining to efficiency. We, however, would count "data retrieval" as another topic of this sentence, and hence as an implied goal specified by the user.

To collect a list of goals implied by the experts, we used the experts' explanations of the relevance of each goal they had specified. It was possible, and even likely, that the terminology the experts were using would not come directly from the pattern. We would need to understand the context of the goal to be able to identify the corresponding goal in the design pattern text. For instance, when asked to describe the goals related to the CACHING pattern, the experts responded with the statement *Performance: Avoid unnecessary resource acquisitions*. We split this into two goals: the explicit goal, performance, and the implied goal, avoidance of resource acquisition. Thus, we collected both the set of goals explicitly specified by the experts, and also a list of implied goals.

We compared the results in two ways. First, we counted exact matches of the goals specified by both the expert and by the user. The goals had to match not
only in terminology, such as “efficiency”, but in purpose and intent, as in “efficiency for the sake of”. Then we counted the number of matches of implied goals listed both by the experts and the DPRG tool user.

We also noted the number of queries the user performed to collect the list of goals.

5.1.2 Results

We now present the results of the completeness case study. We begin with a discussion of the study validity. Then we present results of the analysis for the three patterns used in this study. Finally, we present results of the case study in general.

Validity

To address external validity,\(^1\) we chose three patterns that differed in authorship and size. This variety increases the likelihood that the results of this study may apply to other patterns.

To ensure construct validity,\(^2\) we imposed restrictions on the actions performed by the DPRG tool user. First, we stipulated that only the node expansion technique could be applied. Second, only upwards queries could be performed. Third, the user was required to start with a graph of the link-level, and could query upwards only from source-level nodes. These restrictions ensured that the user could not “fish” for design goals based on prior knowledge or intuition, but was forced to arrive at them in a reproducible manner. For example, the tool user was not allowed to perform a regular expression search for efficiency although it would likely be a design goal in the CACHING pattern. To further ensure construct validity, we did not inform the user of the goals identified by the experts, and ensured that the user had not read the design patterns used in the study.

\(^1\)External validity refers to the generalizability of the results of a study
\(^2\)Construct validity refers to whether a study effectively measures what it is intended to measure
The DPRGs involved in the study were created by the investigator before the developers had identified the design goals, hence keywords pertaining to the goals were not necessarily in the dictionary, and pattern-level nodes connected with the design goals were not necessarily linked to source-level nodes. Still, the choice of dictionary keywords was identified as a threat to construct validity. The creator of the DPRGs had experience in constructing dictionaries for creation of the pattern-level graph; and so was more likely to identify keywords related to design goals. This, however, does not guarantee that a particular design goal would be found by the DPRG tool user. The stipulation that the user could not perform regular expression searches ensured that the user could not scan the DPRG for concepts, and then trace down to portions of code from them. Inclusion of a concept in the dictionary does not ensure that the description of this concept would be connected to relevant portions of the graph: that is essentially left up to the original author of the text. It will require further study to determine how well DPRG readers are able to identify concepts related to a portion of code if those concepts were not included in the dictionary. Since this was not a central issue in the validation of the thesis claim, we postpone this investigation until a later time.

Analysis of Caching

The design goals related to CACHING identified by the experts and by the user are listed in Table 5.2. In the first column, the design goals are listed. The experts column shows that two of these goals (performance and efficient lookups) were explicitly specified, and one goal, the avoidance of resource re-acquisition, was implied.

The user examined the DPRG for the CACHING pattern, and arrived at the list of goals shown in the User column. Three goals were identified, and two were implied.

The goals API simplicity and encapsulation of functionality were identified by the user, but not by the experts. When asked to review these goals, the experts
Table 5.2: Design goals Identified for the CACHING Pattern

<table>
<thead>
<tr>
<th>Goal</th>
<th>Experts</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>✧</td>
<td>✧</td>
</tr>
<tr>
<td>Avoid resource re-acquisition</td>
<td>◊</td>
<td>◊</td>
</tr>
<tr>
<td>API simplicity</td>
<td>□</td>
<td>✧</td>
</tr>
<tr>
<td>Encapsulation of functionality</td>
<td>□</td>
<td>✧</td>
</tr>
<tr>
<td>Efficient lookups</td>
<td>✧</td>
<td>◊</td>
</tr>
</tbody>
</table>

specified-✧ implied-◊ accepted-□ rejected-■

accepted them as relevant to this implementation.

The set of goals were self-reported by the DPRG tool user. The user also identified the supporting sentence chains for each goal. The user applied three node expansion queries, and identified ten sentences pertaining to design goals out of 99 sentences in the CACHING DPRG.

Analysis of Thread-Specific Storage

Table 5.3 presents the design goals given by the experts and user for the THREAD-SPECIFIC STORAGE pattern.

In the case of this pattern, the experts identified two goals, avoidance of race conditions, and avoidance of blocking overhead. The user explicitly identified four goals, none of which were originally identified by the experts, and implied the avoidance of race conditions goal.

The user missed the blocking overhead goal. This omission could be due to several factors. First, blocking is not mentioned specifically in the pattern, though locking is mentioned extensively. Also, in the DPRG used for this investigation, none of the pattern-level entities connected explicitly to locking were linked. For instance, one pattern-level class related to locking; this class was not linked to the
Table 5.3: Design goals Identified for the THREAD-SPECIFIC STORAGE Pattern

<table>
<thead>
<tr>
<th>Goal</th>
<th>Experts</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avoid race conditions</td>
<td>●</td>
<td>◊</td>
</tr>
<tr>
<td>Avoid blocking overhead</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Correctness</td>
<td>□</td>
<td>●</td>
</tr>
<tr>
<td>Flexibility</td>
<td>□</td>
<td>●</td>
</tr>
<tr>
<td>Reusability</td>
<td>□</td>
<td>●</td>
</tr>
<tr>
<td>Maintainability</td>
<td>□</td>
<td>●</td>
</tr>
</tbody>
</table>

specified-● implied-◊ accepted-□ rejected-■

source-level. However, a member method of that pattern-level class was linked to the source-level. The pattern-level entity linked to the member method, however, was not explicitly mentioned in connection with locking. Due to the stipulation that the user not perform queries from un-linked nodes, links from the goals related to locking were not encountered.

The experts reviewed the list of additional goals identified by the user and accepted them as applicable.

The user applied four queries, and identified five sentences related to design goals out of 214 in the THREAD-SPECIFIC STORAGE DPRG.

Analysis of Strategy

The experts explicitly identified two goals for the STRATEGY pattern (Table 5.4): reusing strategy code and keeping interaction with strategized code to a minimum for the sake of reuse. The experts also implied maintainability of code as a goal related to this pattern.

The user identified all of these goals, and three others: efficiency, minimized coupling, and code understandability. When the experts reviewed these goals they
Table 5.4: Design Goals Identified for the Strategy Pattern

<table>
<thead>
<tr>
<th>Goal</th>
<th>Experts</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reuse code for different purposes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimum interaction with strategy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Efficiency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimize coupling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code understandability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code maintainability</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

specified-♦ implied-◊ accepted-☐ rejected-■

The user applied six queries and identified seven sentences to collect the set of goals.

Analysis of Completeness

From these results, we conclude that the DPRG representation and tool do enable a developer to effectively navigate to the relevant design goals for a portion of a code base. We derive this from the fact that the user located all but one of the goals mentioned by the developers, and that none of the additional goals located by the user were rejected by the experts. It is not the case that all goals in the design patterns were relevant. The experts reported that there were goals in the design patterns that were not relevant to the Zen code base. These irrelevant goals were not reported by the user. We consider the one goal missed by the user in the Thread-Specific Storage case as falling within an acceptable level to be able to make this claim. We believe that the restrictions on the investigation style of the DPRG tool user were responsible for the failure to identify the design goal.
5.2 Case Study: Confidence

To investigate whether a developer with access to a DPRG for part of a code base is more confident about how that code base achieves certain design goals, we conducted a study with two cases.

5.2.1 Design

In this section we will describe the design of the confidence case study. We describe the subjects in the study, the study preparation, the format of the cases, and the data collected.

Subjects

The subjects of the case study were developers from Siemens AG. To be eligible for this study, the subjects were required to have recently completed tasks that required investigation of code related to a design pattern.

Prior to the Session

Each subject provided a distinct code base and indicated a design pattern of interest relevant to that code base. The experimenter provided a DPRG for the chosen pattern, and a source-level graph for the code base. The DPRGs were created without in-depth knowledge of the pattern itself. The experimenter created dictionaries of keywords for each pattern, and added the relevant sequence annotations based on where in the text the word "then" was used. The source-level graphs were created by running the code provided by the participants through the Chava[40] program analysis tool, and then using part of the DPRG tool to post-process the output. We chose to provide the subjects with these artifacts to reduce their time commitment for the study to one hour each, plus one hour of training and debriefing time.
Case Format

In each study, the subject, a developer from Siemens AG, was asked to use a DPRG to investigate a portion of a system on which they had recently finished a task. The subjects were only given the DPRG related items: They were not allowed to consult their code bases or the design pattern during the session.

Each session was observed by an investigator, who videotaped the session, and asked the subjects status questions, such as “how are you doing”, or “what are you doing”, in order to ascertain the subjects’ status more accurately than just through observation. The subjects were asked to talk through their work [66]. Each case consisted of briefing, training, task, and de-briefing phases.

Briefing. We informed the subjects that the study was to observe their use of the tool when investigating design goals. We chose this approach, rather than asking the subjects directly about their confidence, to avoid leading them to a positive response, and to avoid giving them an impression of a narrow definition of the term confidence.

Training. Subjects were introduced to the DPRG tool through a short training exercise. First, they were shown how to read the pattern-level graphs. Then, they were asked to perform a simple exercise of linking a source-level to a pattern-level graph, and then to perform a query on the DPRG. The pattern chosen was VISITOR, and the training system was a toy implementation of that pattern.

Pre-questioning. After the subjects were trained to use the DPRG tool, the investigator discussed with them the design goals they encountered while working previously with their code bases. The point of this interview was to determine two things: which goals to investigate during the treatment, and how well the subjects were aware of how the goals played out in their code bases.

Treatment. Each subject was asked to use the DPRG tool to investigate the goals they had described during the pre-questioning. Each subject was told they had “around an hour” to do this. The time constraints were not fixed because the
sessions were conducted during the subjects' regular working hours, and had to be able to be terminated if so required. We allowed a subject to lengthen the sessions if they desired. During a session, the investigator asked the subject what they were doing, and why, in order to better record the subjects' progress. A subject was permitted to ask for technical assistance such as help with file locations, or assistance in window scrolling, but not for advice about the investigation itself.

De-briefing. After a subject decided they were finished with their investigation, the investigator engaged them in a free-form discussion about their experience and opinions on the DPRG tool. These questions were not intended to address the issue of confidence with relation to design goals, but rather to gather feedback on DPRG tool usability. The investigator asked several questions:

- Did you find this approach helpful or frustrating? How?
- How would you have undertaken such an investigation without this tool?
- How does this compare to the investigation you conducted originally for your task?

Rather than report the results of this de-briefing in the analysis of completeness, we discuss the main findings from these questions in Chapter 6.

Data Collected

The treatment portion of each session was videotaped and then transcribed. The data collected included:

- the queries performed,
- the subjects' running commentary through the treatment,
- the subjects' responses to the pre-questioning and the de-briefing, and,
• the level at which each query was performed, and at what time in the course of the investigation.

To obtain data for analysis, several passes over the transcripts were made:

1. Formation of the transcripts themselves from the video recordings
2. Annotation of the transcripts with the log of the queries performed
3. Highlighting places in the transcripts where the participants pointed out that they had done something in error, including made an assumption, or performed a query, or made a link.
4. Highlighting places in the transcripts where the participants pointed out that they had gained knowledge, specifically points at which they stated "I had not known this before", or "this is new to me".

Analysis of the data then consisted of corroborating the knowledge gained by the participants with the log of the queries performed to ensure that the queries performed provided the information said to have been gained.

We classified each query performed on a seven point scale:

1. Source-level node expansion on a node not included in the link-level
2. Source-level node expansion on a node that is included in the link-level
3. Link-inference
4. Pattern-level node expansion on a node that is included in the pattern-level
5. Pattern-level entity expansion on a node that is not included in the link-level, but which is a participant in the design pattern
6. Pattern-level node expansion on an entity that is not included in the link-level, and that is not, strictly speaking, a single participant in the design pattern (such as the registration interface node, for instance)
7. Pattern-level regular expression on a high-level concept (such as “efficiency”)

These classifications were used to trace the participants’ paths through the DPRG during a study session. These are depicted in Figures 5.1, and 5.2.

5.2.2 Case 1

The first subject was working on a portion of a system that supported the visualization of object connections in a distributed system. The subject needed to implement a locking mechanism to allow only one method to execute in an object at a time. The locking mechanism implemented consisted of a loop to check the accessing object and method identifiers. The subject later learned about the MONITOR OBJECT pattern [11], which synchronizes concurrent method execution within an object, ensuring only one method runs at any given time within the object. The subject believed that this design pattern closely represented her solution to this problem. This subject investigated the MONITOR OBJECT pattern’s correspondence to the code base with limited success, and wished to revisit the investigation in this study.

The subject mentioned three goals of interest: locking, blocking, and synchronization.

Actions

The subject performed six investigative actions in the study session. Figure 5.1 depicts the levels at which these actions were taken. In this figure, each grey area encloses one action as described in the list to follow. Actions may contain more than one query, and so are not equal in width in the Figure. The vertices represent individual expansion or regular expression queries, and the lines connecting them are included to show continuity. The dashed line between actions three and four shows that action four was a new avenue of investigation. Solid lines link actions that are part of one investigation.
Figure 5.1: Subject 1: path through the DPRG of the MONITOR OBJECT pattern. Grey areas represent actions, vertices represent individual queries. Lines connecting queries show continuity of an investigation. A dashed line represents the start of a new line of investigation. The height of a vertex represents the concreteness of the query: a high-vertex is an abstract query, a low-vertex is a concrete query. Vertices shown at in the seed-file portion of the link-level indicate link inspections.
1. The subject began by positing three seeded links, including an association between a class, *Monitor Condition*, in the pattern, and a method, *TransmitCallID*, in the code base.

2. The subject then used the DPRG tool to infer more matches, resulting in the tool suggesting that the *TransmitCallID* method was a possible match for the *synchronized method* pattern element.

3. The subject inspected the subgraphs surrounding the linked pair of the *synchronized method* and the *TransmitCallID* method by looking at the link-view for that link.

4. The subject began to query at the pattern-level, searching for *block*, which matched 18 sentences. One of these sentences included the phrase: “if a synchronized method must block or cannot make immediate progress, it can wait on one of its monitor conditions”.

5. The subject then revisited the view of the link between the *synchronized method*, and the *TransmitCallID* method, and then expanded the *synchronized method* node at the pattern-level. These actions resulted in a large graph that included information about the relationship between the *synchronized method* and the *monitor object*. Specifically, a graph containing the following sentence was displayed: “A monitor object therefore contains a monitor lock that serializes the execution of its synchronized methods, as well as one or more monitor conditions used to schedule the execution of synchronized methods within a monitor object”. At this point, the subject reported that her initial seeding of the *TransmitCallID* method with the *monitor object* class was incorrect, and that the proposed link from the *synchronized method* to the *TransmitCallID* method was correct.

6. Within the same view window, the subject then expanded the *monitor object* at the source-level, revealing the caller of the *synchronized method* to be the
client. After this, the subject continued to browse the view window, and noted that the methods exported by the monitor object, are often synchronized.

The subject then stated that she believed the code matched the MONITOR OBJECT pattern.

Analysis of Case 1

Before using the DPRG, the subject was unsure whether the code implemented the design pattern. The subject's first action, where an initial seed was made incorrectly, suggests that the subject did not have a good understanding of the pattern. Before accepting the tool's suggestion on a different link between the elements in question, the subject investigated the design context for various portions of the pattern, including, in the fifth step above, expanding the synchronized method at the pattern-level. After the final action, the subject stated that she believed her implementation was structurally similar to the solution described in the pattern. Furthermore, the subject was able to articulate that her solution shared the same design goals of synchronizing methods, and allowing only one synchronized method in an object to run at a time to prevent race conditions, with the pattern. The subject was also able to articulate that the code and the pattern shared the same drawback: it may be difficult to change the synchronization policy once it is determined and coded.

5.2.3 Case 2

The second subject was instrumenting methods in a system implementing certain design patterns in order to generate sequence diagrams from system execution traces. This study involved the FORWARDER-RECEIVER pattern, which consists of a forwarder, who upon receipt of a message, sends the message on to a receiver. The subject reported only a cursory understanding of the FORWARDER-RECEIVER pattern at the start of the study. She reported being familiar with the code base. The
Figure 5.2: Subject 2: path through the DPRG of the FORWARDER-RECEIVER pattern. The height of a vertex represents the concreteness of the query: a high-vertex is an abstract query, a low-verted is a concrete query. Vertices shown at in the seed-file portion of the link-level indicate link inspections.

developer noted no design goals of specific interest.

**Actions**

The subject performed six major actions. The level at which each of these actions was taken is depicted in Figure 5.2, which has the same format as Figure 5.1.

1. The subject posited one seed, the class representing the *forwarder*, and then used the tool to infer more links. The tool suggested that two methods in the
pattern, the *marshal* method, which prepares a method for forwarding, and the *sendMsg* method, which performs the action, should both be linked to the *sendCmdAsynch* and *sendCmdSynch* methods in the code.

2. The subject did not recognize the role of the *marshal* method, and thus chose to see the link details in a separate DPRG view window.

3. The subject then expanded the *marshal* method at the pattern-level. Upon reading the displayed description, the subject noted that it was likely that this functionality had been absorbed into the *sendCmdAsynch* method and *sendCmdSynch* methods.

4. The subject then revisited two source level methods shown in the view window: *sendCmdSynch* and *sendCmdAsynch* and was reminded that the implementation she was instrumenting had taken two approaches to the pattern implementation. These two methods were both linked to the *marshal*. She indicated upon seeing this that she was unsure of the design pattern’s intent with relation to asynchronous versus synchronous approaches.

5. The subject decided to look into these approaches further, and so performed a regular expression search for “asynch”. This search resulted in several sentences, including one stating that the developer “must decide whether the receiver should block until the message arrives”. The subject indicated that she had not previously known that the pattern had not required both the asynchronous and synchronous schemes, but now understood that the original developers of the code base associated with her task had implemented two options for the pattern’s implementation.

6. The subject then began a new DPRG view search for the regular expression *blocking*. This resulted in several sentences, including one that stated that “if the underlying IPC mechanism does not support non-blocking, the developer could use a separate thread to handle communication”. The resultant
The subject indicated that this must have been what was done in the asynchronous version of the implementation.

Analysis of Case 2

Several times during the actions of using the DPRG tool, the subject stated having gained knowledge. Through the use of the DPRG tool, the subject learned, in actions 2 and 5, about differences between the concrete design provided in the pattern and the implementation of the pattern in the system being studied. The subject used information in the pattern-level to investigate the differences, until she was satisfied the differences did not change the intent of the pattern.

The subject also learned, in action 4, about options in the pattern of which she was unaware. With the help of the DPRG tool, she was able to focus on the pertinent parts of the pattern, and was then able to explain the code.

5.2.4 Results

We now present the synthesized results of both cases in this study. We begin with a discussion of study validity, then present the analysis of the results of the claim of confidence.

Validity

We believe our study to have internal validity, due to our analysis methods.\footnote{Internal validity refers to the truth about inferences made from observations in the study, and evaluates how well a cause-effect relationship can be determined between the study treatment, and the observed outcome} We examined the subjects' use of the tool, and their running commentary in the sessions, indicating what they had learned about their code base. If the developer reported
learning something about the goals of their code relevant to their initial investigation
task we consider their confidence increased.

To address construct validity, and to remove the possibility of hypothesis
guessing\textsuperscript{4} and evaluation apprehension\textsuperscript{5} [75], we did not inform either subject of
the true intent of the study, telling them that we wished to observe their use of the
DPRG tool when investigating design goals as opposed to informing them that we
were examining their confidence with relation to design goals. We chose to focus on
recently completed tasks because the developers would be familiar with the design
goals relevant to the code. One potential drawback of this study format is that
a developer might be hesitant to admit that they were unaware of some design
goals when performing the original task. This stipulation may also have weakened
our results since all developers had performed investigations that they felt were
sufficient to their original tasks. If their work had been thorough, there would be
few unknowns with relation to the goals which were relevant, or how they played
out in the code base.

We do not believe that the additional time spent was a factor in the subjects’
performance. Both subjects had previously undertaken these investigations, and had
spent several hours doing so. The additional hour spent using the DPRG tool for
this analysis does not significantly increase their time investment.

We also do not believe that pressure to perform well was the reason for
the subjects’ positive outcomes. There are three reasons we believe this is not the
case. First, subjects were able to choose the actions they performed, and gave
their reasoning for their choices at the beginning of each activity. In both cases,
the subject chose investigations which were of interest to them, and which they
noted as being relevant to their original task. The subjects did not know which
\textsuperscript{4}Hypothesis guessing refers to a situation in which a subject becomes aware of the hypothesis
of the study. This is a threat to construct validity since this could prompt a subject to alter their
behaviour to either support or refute the hypothesis
\textsuperscript{5}Evaluation apprehension refers to the situation in which a subject suffers from anxiety due
to the fact that they are being observed. This anxiety is a threat to construct validity since the
subject may wish to “look good” or “do their best” in the study setting
investigation activities would result in new information. It was not the case that every query resulted in new knowledge about their code base or the pattern they were investigating. Had they felt pressure to perform well, this would likely have only affected the number of queries but not the outcome of those investigations. Second, the subjects were able to terminate the investigation at any time. In both cases, subjects decided when they wished to abandon a path of investigation. Third, subjects had both attempted these investigation tasks in their work prior to the study. The subjects had reported attempts to relate their code to a design pattern. These attempts had resulted in varying levels of success. Had the subjects performed their investigation with the same methods they used originally, while in a study setting, it is likely that their levels of success would not have changed. The subject from case 1 made remarks which support this theory:

I didn't even really understand which part of my code is really a condition and which is the synchronize method. I didn't know this before. [The DPRG] helps because you really have a crosscutting view, you can read it in the sentence, and you can match this to the method you see in the code. But [the pattern] is quite difficult to understand so it helps to have this [pattern-level] view and this matching into the source code.

Since we examined only two cases, the external validity of our study is limited. However, we believe this study shows promise due to its high degree of realism. First, subjects were software developers from industry. Second, subjects were revisiting an investigation task they had previously performed as part of their work as software developers. Third, they were performing this task on an industrial code base. Fourth, the two cases differed in style: one subject was familiar with their code, yet unable to conclude how well it resembled the design of a particular pattern, the second subject was aware that the code was intended to implement the pattern closely, and had performed prior investigation that she had deemed sufficient to verify this constraint.
Analysis of Confidence

We evaluated confidence in terms of whether the subject reported new knowledge that would positively affect their task, and whether this new knowledge was relevant to their original task. Both subjects displayed increased knowledge of how design goals were carried out, either at the pattern-level (as in Case 1) or at both the pattern-level and source-level (in Case 2).

5.3 Case Study: DPRG Tool Lightweightness

The DPRG approach is intended to enable developers to obtain information about design goals with a small investment of time and effort.

To determine whether the DPRG tool accomplished this goal, we set up a study with two cases in which developers from industry were asked to find portions of code relating to the goal of efficiency in the OBSERVER pattern as implemented in the JHotDraw code base. The subjects were assigned a task to simulate a software evolution situation. To accomplish this task, the subjects were required to create the pattern-level graph, which involved creation of the dictionary for the design pattern, and the link-level graph of the DPRG.

5.3.1 Design

This section describes the design of the lightweightness case study. First, we discuss the subjects in the study; second we describe the preparation for the study; third, we detail the code base used; fourth, we outline the format of the cases; and finally we specify the data collected.

Subjects

The two subjects in this study were developers at Siemens AG. Each subject had over three years of experience as a software engineer. To be eligible for this study, the
subjects had to have a basic, though not intimate, familiarity with the OBSERVER pattern. We determined that the subjects met this criteria by asking them how much exposure they had to the OBSERVER pattern. Both subjects reported that they had read but not implemented the OBSERVER pattern.

**Prior to the study**

The investigator provided the subjects with a sequence annotated text of the OBSERVER pattern, and a source-level graph of the JHotDraw code base. These artifacts were provided since their creation is not facilitated by the DPRG tool although they are necessary to use it. We wished to focus on how the subjects fared with the DPRG tool itself, rather than the setup procedure. We also believe that the setup procedure could be automated in future implementations of the DPRG tool.

**JHotDraw Code base**

JHotDraw is a Java framework for graphical editors. The designers of JHotDraw aggressively used design patterns in its construction. JHotDraw is comprised of approximately 170 classes, and approximately 26,000 lines of commented code including sample applications.

**Case Format**

The subjects in this study were asked to form a DPRG of the OBSERVER pattern for the JHotDraw code base. This involved creating a dictionary for the OBSERVER pattern, and linking the pattern-level graph to the source-level of JHotDraw. In order to ensure that we could observe the subjects using the DPRG tool, the subjects were not allowed to consult either the JHotDraw code, or the design pattern itself, during a session.

Each case was observed by the investigator, who videotaped the session, and who asked status questions of the subject in order to ascertain their actions more
explicitly than just through observation. The subjects were asked to talk through their work [66]. Each case consisted of briefing, training, and task phases.

**Briefing.** The investigator informed the subjects that the objective of the study was to observe their use of the DPRG tool while performing a task. They were not informed that we were examining the lightweightness of the tool.

**Training.** Subjects were introduced to the DPRG tool through a short training exercise. First, each subject was shown how to read the pattern-level graphs. Then, each subject was asked to perform a simple exercise of linking a source model to a pattern-level graph, and then to perform a query on the DPRG. The pattern used in this exercise was the *VISITOR* pattern, and the training system was a small implementation of that pattern.

**Task.** Subjects were asked to spend an hour forming a DPRG of *OBSERVER* and JHotDraw and investigating the concept of “efficiency” in *OBSERVER*. Efficiency is discussed twice in the *OBSERVER* pattern. In particular, we informed the subjects that for task to be considered completed successfully they must have identified portions of the pattern-level related to one discussion of efficiency, and identified how those portions relate to the source-level graph. We wished to give a specific task to provide a basis for measuring the quality of the subjects’ work. We constrained the task to examining just one description of efficiency since this was tractable within the allotted time. Were a developer to examine efficiency in its entirety, more time than the subjects were able to spend may be required.

**Data Collected**

To analyze how the subjects fared with the DPRG tool we collected several data:

- the queries performed,
- the subjects’ running commentary through the treatment,
- the subjects’ responses to questions about their experience with the DPRG
tool

- the time taken to form a dictionary for Observer,
- the time taken to link the pattern- and source-levels,
- the DPRG level at which each query was performed, and
- at what time each query was performed in the course of the task.

We analyzed the subjects' responses and running commentary to determine where the subjects were having trouble using the tool, and what information they were gathering from the queries they performed.

5.3.2 Case 1

Subject 1 performed seven actions over 70 minutes. These actions included two pattern-level queries, two source-level queries, and iterative linking. By the end of the session, the subject had successfully completed the task. The actions taken by the participant are presented in Table 5.5.

5.3.3 Case 2

Subject 2 performed eight actions over 120 minutes. These actions included two pattern-level queries, and a significant amount of source-level graph browsing. By the end of this session, this subject had not fully completed the task, though was successful for a major portion of the task. The actions performed by subject 2 are described in Table 5.6.

5.3.4 Results

We now present the results of the case study. We begin with a discussion of study validity, and consider factors which may have affected the results of the study. Then we provide analysis of the two cases.
Table 5.5: Actions of the Lightweightness Subject 1

<table>
<thead>
<tr>
<th>Action</th>
<th>Duration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10 mins</td>
<td>Dictionary creation</td>
</tr>
<tr>
<td>2</td>
<td>10 mins</td>
<td>Pattern-level regular expression search for “efficiency”. This revealed information that the registration interface has an effect on efficiency of the Observer pattern.</td>
</tr>
<tr>
<td>3</td>
<td>10 mins</td>
<td>Pattern-level regular expression search for “regist”. This revealed the pattern-level node attach method as the main actor in the registration interface</td>
</tr>
<tr>
<td>4</td>
<td>5 mins</td>
<td>Source-level search for the term “subject”. This produced no results.</td>
</tr>
<tr>
<td>5</td>
<td>3 mins</td>
<td>Source-level search for the term “observer” followed by browsing the source-level graph. This revealed the class abstractFigure.</td>
</tr>
<tr>
<td>6</td>
<td>2 mins</td>
<td>The subject seeded in abstractFigure for the pattern-level node subject, and asks the tool to infer more matches. This is the only time this feature was used.</td>
</tr>
<tr>
<td>7</td>
<td>30 mins</td>
<td>The subject continued seeding and viewing links until through source-level graph browsing, the subject found the FigureChangeListener to be the observer, and addFigureChangeListener to be the attach method mentioned during the earlier pattern-level search from action 3.</td>
</tr>
</tbody>
</table>
Table 5.6: Actions of the Lightweightness Subject 2

<table>
<thead>
<tr>
<th>Action</th>
<th>Duration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15 mins</td>
<td>Dictionary creation</td>
</tr>
<tr>
<td>2</td>
<td>5 mins</td>
<td>Looking at the source-level file. Since this activity was not commented on by the subject, we are unable to determine the goal of the activity.</td>
</tr>
<tr>
<td>3</td>
<td>5 mins</td>
<td>The subject noticed the class <code>eventListener</code> and explored more in the source-level about that class.</td>
</tr>
<tr>
<td>4</td>
<td>5 mins</td>
<td>Seeds the <code>eventListener</code> class for the <code>observer</code> and infers matches.</td>
</tr>
<tr>
<td>5</td>
<td>60 mins</td>
<td>The subject continues source-level graph browsing and using the link inference feature of the tool until links have been suggested for all portions of the pattern except the <code>attach</code> operation. The subject decides to move on and look at the pattern-level to see whether this linkage is enough.</td>
</tr>
<tr>
<td>6</td>
<td>5 mins</td>
<td>Pattern-level search for &quot;efficiency&quot; revealing the <code>registration</code> interface.</td>
</tr>
<tr>
<td>7</td>
<td>3 mins</td>
<td>Pattern-level search for &quot;reg&quot;, revealing the <code>attach</code> method as central to efficiency. Based on this information, the subject returns to the linking process to attempt to find the <code>attach</code> method.</td>
</tr>
<tr>
<td>8</td>
<td>22 mins</td>
<td>The subject posits the <code>DrawingInvalidated</code> method for the <code>attach</code> method, but after more source-level graph browsing determines that this could not be correct. The subject continues examining the source model and asks the tool to infer more links.</td>
</tr>
</tbody>
</table>
Validity

To ensure construct validity, we did not provide the subjects access to the design pattern or the JHotDraw source: Subjects were only provided the DPRG tool. The investigator gave no advice or encouragement that would aid in the task. These measures helped ensure that all progress made by the subjects could be attributed to the tool.

The subjects were given up to two hours to complete the task. A subject was not hurried in their attempts. A shorter time constraint may have lead to falsely improved timing results.

The preliminary nature of the DPRG tool at the time of the study may have affected the results of the study, since it had several limitations. First, the subjects could not query from within the link inspection view. To explore the context of a link more extensively than that provided by the inspection view, the subjects had to save the view to a file, and then re-load it from within the query view. Only one subject chose to do one such query. This limitation may have hurt the subjects' abilities to efficiently form links.

Second, the format in which seeds were specified was more restrictive than presented in Section 4.2.2, and more restrictive than in the other studies. The subjects required time to insert the correct spacing and quotation marks in the seeds so that they would be accepted by the tool.

Third, the distribution of the GraphViz [41] tool was problematic. At times, GraphViz required up to 20 seconds to load the results of a query. Intermittently, GraphViz was unable to load a DPRG view, causing minor delays while the tools were re-started. This lag time may have affected the approach the developers took to investigating the DPRG.

Since the sample size of this study is small, and the task contrived, the external validity is limited. To obtain some generalizability, we chose subjects who were representative of developers in many ways: they had an average understanding of
the Observer pattern, and had exposure to, but not immersion in, design patterns in general. Each subject had approximately three years of software development experience. We also chose a task that, while contrived, was not trivial. JHotDraw is a non-trivial code base and since the developers were not given the original code, they were not able to read the comments in the code indicating where patterns were applied.

**Analysis of Lightweightness**

The results for both subjects are summarized in Table 5.7. Both subjects were able to create the dictionary and DPRG for the Observer pattern with relative ease: Subject 1 took 10 minutes to perform this task, and Subject 2 required 15 minutes. The dictionaries created by the two subjects were identical within two trivial terms: one subject included the term `state`, and the other included the term `storage`. The creation of the link-level, and the search for efficiency, were integrated into one main activity, since these tasks could be undertaken in any order.

Subject 1 was successful in the task, finding the connection from efficiency to the subject's registration interface, where observers register for particular events, through to the `attach method`, which allows registration, and finally to the source-level node `addFigureChangeListener`, which corresponds to the `attach method`. This
subject accomplished this in 70 of the 120 allotted minutes.

The second subject was able to achieve most of this goal. Subject 2 also chose to trace the registration interface through to the source-level. After the full 120 minutes, this subject had correctly identified an observer, in the JHotDraw code base, and had identified its associated update method. The subject had also identified a subject in the code base. However, within the allotted time, the subject was unable to identify the source-level method corresponding to the attach method.

When examining the actions of both subjects, and in particular those responsible for the non-completion of Subject 2, it was evident that at the time of the study, the mechanisms for browsing the code base were inadequate. While subjects were able to view the subgraphs surrounding links, it was more difficult for them to browse the source-level freely. This led Subject 2 to spend considerable time reading the source-level file itself, which is in database form, and was not intended for this purpose. In contrast, Subject 1 chose to work around this problem by positing seeds, and then using the link inspection feature to produce limited views of the code. This approach appeared more successful than the browsing approach of Subject 2. Additionally, Subject 2 seemed unclear on how the DPRG tool proposed suggested links, and spent time considering each one. Though it was clear to the subject at first glance that a link was incorrect, the subject was under the initial impression that the links suggested would necessarily be correct, rather than just suggestions. After grasping this fact, the subject was able to make forward progress. Subject 1 displayed understanding of the nature of link proposal, which could explain the faster time, and better success. The investigation paths chosen by the two subjects also differed. Subject 1 began exploring at the pattern-level, and thus determined early on which portions of the pattern were relevant to efficiency. Subject 2 began by linking, and was not as aware which portions of the pattern structure would be relevant to the task of finding code related to efficiency. Subject 1 also spent more time examining the pattern-level than Subject 2, though they arrived at the same
query results. We are unable, however, to determine with certainty whether the differing strategies affected subject performance.

Based on the progress made by the subjects in the allotted time, we conclude that the DPRG tool can be applied in a lightweight manner within the context of a task. However, usability issues, including enhanced browsing of the source-level graph, and more explicit indication of the nature of link possibilities, would likely enhance its effectiveness. These results are corroborated by the results from the study on developer confidence (Section 5.2) in that developers in that study were also able to apply the linking and querying portions of the DPRG approach and tool within an hour.

5.4 Comparative Experiment: Pattern-Level Efficacy

We conducted a comparative experiment [4] to determine whether a graphical representation of design pattern text, as provided by a pattern-level DPRG, would facilitate identification of design detail and context associated with concepts and structural entities described in the design pattern. We noticed that developers working only from a design pattern were frequently incorrect or incomplete about their descriptions of entities and concepts. The information related to these entities and concepts crosscut the pattern description. The developers given the DPRG representation were able to report the detail and context missed by the other study subjects.

Half of this study was described in Chapter 2, in which we focused on the performance of the control group. Here, we draw a comparison between the group described in Chapter 2, and a group of developers given a pattern-level graph of a DPRG.
5.4.1 Design

This section describes the design of the study. We begin by discussing the study format, then we describe the patterns used as a basis for the experiment, then the subjects in the study, the experimental preparation, and the evaluation questions.

Format

We broke the study into two blocks, each of which consisted of four, single subject trials. In each block, half of the subjects, the test group, had access to a DPRG of the design pattern; the other half, the control group, worked only from the pattern. All subjects were asked questions about the design pattern. We compared the responses of the control group to the test group within each block. We then compared the results between the blocks.

The subjects in the first block were software developers from Siemens AG. These subjects worked with the REACTOR pattern [61]. The subjects in the second block were four graduate students from the University of British Columbia (UBC). These subjects worked with the VISITOR pattern [23].

Patterns Used

We used two different patterns to help reduce the likelihood that a problem in understanding the pattern was related to the way in which the pattern was written, or to the questions we asked. The two patterns we used have different authors and are of differing size: the VISITOR pattern is short but subtle; the REACTOR pattern is longer and more detailed. Refer to Table 5.1 for the sizes and descriptions of each of these patterns.

Subjects

We kept the skill set of subjects within each block as similar as possible. All of the subjects in the REACTOR experiment were non-native English speakers, with similar
experience in reading and writing English. Each subject possessed the equivalent of a Bachelor’s degree in Computer Science, and had at least one year of experience working with Java in an industrial setting. Each subject was screened to be familiar with design patterns, but no exposure to the REACTOR pattern.

The subjects in the VISITOR pattern experiment were all PhD students at the University of British Columbia. None of the subjects had previous exposure to design patterns. Subjects had programming knowledge but no exposure to large industrial code bases. All reported working knowledge of object-oriented concepts. Each subject was a native English speaker.

**Experimental Set-up**

In each trial, a subject was given the same set amount of time to read a hard-copy of the assigned design pattern. At Siemens, the subjects were given one hour to read the REACTOR pattern. At UBC the subjects were given 20 minutes to read the VISITOR pattern.

After reading the pattern, subjects in DPRG trials were asked to put away their copy of the pattern. They were then given a 20-minute tutorial on reading a DPRG. After this tutorial, they were asked a predetermined list of questions about the pattern. They were not allowed to refer to the hard-copy of the pattern while answering the questions. They were directed to ask the experimenter to perform an operation on the presented DPRG of the pattern and were able to view the DPRG resulting from the operation. We chose the approach of the experimenter performing the DPRG query because of usability concerns about the DPRG tool interface at the time of the study.

Subjects in the control group were asked the same predetermined questions about the pattern. In contrast to the DPRG trials, these subjects were allowed to refer to their copy of the pattern, and any notes they had taken while reading the pattern.
Subjects in trials involving the VISITOR pattern were asked to answer, as fully as possible, three questions:

1. What allows the VISITOR to directly access the concrete element?

2. How is it determined which operation is executed?

3. What is the sequence of events that occur in the VISITOR pattern?

For the REACTOR study at Siemens, the experimenter was not on-site. Instead, the experimenter interacted with the subjects over the phone and over the web. These subjects were asked a different set of three questions:

1. What does the logging handler register with, and what does it register for?

2. About what does the synchronous event demultiplexer notify the initiation dispatcher?

3. What happens after a connection request arrives?

It was reasonable to expect that both the control trial and the DPRG trial subjects could answer these questions for two reasons. First, the questions we asked of the subjects about the pattern could be answered based solely on the information in the text of the pattern. Second, both the control and the DPRG trial subjects were given ample time to read the pattern, and subjects in the control group were allowed to re-read the pattern as much as they wanted within the allotted time. The control trial subjects were thus not at a disadvantage compared to the DPRG trial subjects.

**Evaluation Questions**

After the subjects had responded to the pattern-specific questions, they were asked follow-up questions.
Subjects in the control trials were asked about their confidence in their answers to the pattern questions, how they used the pattern text to reach their answers, and from where they drew their answers in the pattern text.

Subjects in DPRG trials were asked four questions:

1. Did the graphs help you visualize design entities?
2. Did the graphs help you visualize relationships between entities?
3. Did the graphs help you feel more confident about your answers?
4. Would you choose to use this tool again?

We chose to compare the test group to a control group given a hard-copy of the design pattern. Another possibility would have been to provide the control group with an electronic copy of the design pattern. Had they been provided an electronic version, the control group subjects would have been able to search the text. We chose this approach because we wished to observe and compare against developers' performance using the conventional means of reading a design pattern. As will be shown, results for the control group were the same regardless of whether they were able to identify specific portions of text related to the questions, which suggests that searching would not have been helpful.

5.4.2 Results

This section presents the results of the study. We begin with a discussion of study validity, and then report results with respect to three factors: detail and completeness of responses, willingness to explore the pattern, and reported confidence in responses. Finally, we summarize these results.

Study Validity

The format of the study has several drawbacks, including the small number of subjects, the small number of patterns, and the lack of a group who had both the
pattern text and a DPRG available.

The small number of users and design patterns in our study affects the generalizability of our results. We chose to limit the number of users and patterns because we were focusing on an initial determination of whether the DPRG representation showed promise. We believe our results can answer this question because we varied the background of the subjects, including both experienced software engineers and graduate students, and because we selected patterns that differed in authorship, style, and length.

In our study, we chose to have the test group use only the DPRG rather than both the text of the pattern and the DPRG because we wanted to isolate the use and effectiveness of the DPRG. At this exploratory stage, we did not want to give the subjects a choice about the degree to which they relied on the DPRG to answer their questions.

**Detail and Completeness of responses**

For both patterns, we observed that the DPRG subjects provided highly detailed and precise answers. In contrast, control subjects using the original form of the pattern typically responded with higher-level conceptual information.

The first question in the REACTOR trials, for instance, dealt with a specific example from the pattern. The control trial subjects tended to answer about the general case, rather than about the situation described specifically in the example. Although their answers demonstrated that they understood the relevant concept, they missed stating the precise type of events registered for by the logging handler used in the example, the type of event handler it registered, and details about the entity with which it registered. The DPRG subjects did not miss any of these details.

In the third question of the REACTOR trials, the subjects were asked to explain what happens after a connection request arrives. To help them answer
Figure 5.3: Detail reported by REACTOR subjects. Grey indicates control group responses. All portions were reported by the DPRG group.
the question, the DPRG trial subjects asked to see a graph relating specifically to
the arrival of connection requests. The graph shown in Figure 5.3 was created by
querying for arrival in the context of connection requests, and then by expanding
the sequences in which the arrival nodes appeared.

The answers given by the DPRG trial subjects were more complete and
more detailed than those given by the control trial subjects. Figure 5.3 depicts the
difference in the answers of the two groups. The graph shows the details expressed
by the DPRG subjects. The details given by either of the control trial subjects are
coloured gray. The colouring of nodes shows that the control subjects missed many
design details. Among them, the passive establishment of a sock_stream object, and
the invocation of the synchronously demultiplexing select call.

Willingness to Explore

The DPRG subjects spent more time answering the questions than the control
subjects, who all took less than five minutes to answer each of the questions.

When we asked the control subjects why they did not take more time to
re-read the pattern to provide detail, two of the control subjects said they did not
know it was required, implying that they would not voluntarily do so, and the other
two explicitly said they did not feel they would get anything more out of “flipping
through” the text.

For example, Participant 1, in the VISITOR control trials, responded incor-
rectly to question 1. When asked why he did not look in more detail for the relevant
parts of the pattern, he said:

[it] seemed familiar, but I didn’t think I could flip back and find it. I did
kind of hesitate with the text going [sic] “do I remember at all where
that was, or am I going to have to re-read the whole thing?”, and then
decided I had a pretty good idea where [it] appeared.

In contrast, the DPRG subjects spent approximately half an hour answering
each question. At some point, each of these subjects noted that they believed they had collected all necessary information, but wished to continue exploring “just to be sure”. When asked why, one subject responded:

[With a DPRG] I can start by looking in at a place where I believe is a starting point where I want to begin, and then I can go a little bit out from there, and I can go a little bit down a path and kind of go “no that’s not working out” and quite quickly go another way.

Whereas maybe in the text, it’s more like, maybe I’ll start a paragraph and I won’t know where its going, and I’ll think “I can cut that”, but I feel like maybe I’ve wasted a lot of time, and I SHOULD have read that paragraph. And I feel like [a DPRG] helps me very quickly zoom in on the relationships.

Another indication of the willingness to explore was the inclination of the subjects to modify their original answers based on new information gathered either from the text or from the pattern-level graph. All control subjects were asked to look through the pattern and to report the source of their answers. They all used this perusal to support their original answers, even when those answers were incorrect as happened with two VISITOR subjects. This was the case even in question 3, in which only one portion of the pattern description was needed to correctly answer the question. Both control subjects reviewed this single location, and reported no change to their original answers.

Both during their exploration of the DPRG and upon later reflection, the DPRG subjects all noted that they had incompletely, or incorrectly answered the questions before they began exploration, and that they were able to improve their answers through the use of the DPRGs.
Table 5.8: Confidence and Correctness of Control Group Participant Responses

<table>
<thead>
<tr>
<th>Participant</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>REACTOR Participants</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>confident, generally correct</td>
<td>confident, generally correct</td>
<td>confident, generally correct</td>
</tr>
<tr>
<td>2</td>
<td>less confident, generally correct</td>
<td>less confident, generally correct</td>
<td>less confident, generally correct</td>
</tr>
<tr>
<td><strong>VISITOR Participants</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>less confident, incorrect</td>
<td>confident, incomplete</td>
<td>confident, incomplete</td>
</tr>
<tr>
<td>2</td>
<td>less confident, incorrect</td>
<td>confident, more complete</td>
<td>confident, incomplete</td>
</tr>
</tbody>
</table>

**Level of confidence in answers**

We asked both the control and test subjects to comment on their confidence in their responses. The control trial subjects' responses are summarized in Table 5.8. Although all control trial subjects reported that they could not be fully confident about the completeness of their answers, they were confident about the correctness of their answers. Only one REACTOR control subject admitted little confidence about the correctness of all his answers. The two VISITOR control subjects expressed complete confidence in their answers to questions two and three, but less confidence about question one, although they both strongly believed that they were partially correct. The other REACTOR control subject felt confident about all his answers. To see if they were correct in their levels of confidence, we examined their answers.

As mentioned before, the REACTOR control subjects both answered all the questions mostly correctly, while missing details of the answer.

The control subjects in the VISITOR experiment both rightly lacked confi-
dence on question one, which they both answered incorrectly. For question two, about which they were both highly confident, they answered incompletely, forgetting that it is both the type of the visitor and the type of the concrete element that determine which visit operation is eventually called. One said that it was only the type of the visitor, the other said only the type of the concrete element. In question three, only one subject was able to give details about the invocation of the visit and accept operations; neither recalled how the accept operation was called (even though one was working from the interaction diagram shown in the text, page 335, [23]).

The DPRG subjects all stated that the graphs helped them collect the relevant information together, so they could answer the questions more completely and with more detail. They all noted that they felt more confident answering the questions using the graphs, than they did answering from memory before using the graphs. Three of the four DPRG subjects noted that if they were able to refer both to the text and to the graph, they would feel more confident about their answers.

Analysis of Efficacy

The results of the study indicate positive results in three perspectives: DPRG readability, support for improving the understanding of design concepts, and support for connecting design context to design elements.

After a 20-minute tutorial, all subjects in the DPRG trials were able to read the graphs with relative ease, and were able to collect the information displayed in the graphs to fully answer the questions posed.

The DPRG subjects answered the questions in a more detailed way than those using the pattern text because the DPRG subjects examined portions of a relevant DPRG sub-graph containing the details before answering. The control subjects, in contrast, referred specifically to only one portion of the text per question, and even then, they did not delve deeply enough in the text to draw out all relevant details.
Finally, the DPRG subjects noted design concepts that provided context for the design elements involved in answering the questions. For instance, in the REACTOR trials, only the DPRG subjects noted information about how a process blocks while awaiting arrival of events. This information helps ensure the concept of the responsiveness of servers to clients. In the case of VISITOR only the subjects in the DPRG trials connected the double dispatch concept to how the method to be executed is determined. In each of the DPRG trials, the subject noted the relevant concept information only after seeing it connected to parts of the graph being viewed.

The efficacy of the pattern-level is corroborated by both the results of the study on developer confidence (Section 5.2) and on completeness of reported information (Section 5.1). In the confidence study, developers were able to use the pattern-level portion of the DPRG to obtain relevant information about how their code related to design goals described in the design patterns implemented. Similarly, in the completeness study, the DPRG user was able to use the DPRG tool to identify relevant design information about all of the patterns described by the expert participants.

5.5 Synthesized Results

The investigation paths followed by the subjects in the confidence and lightweightness studies support our motivations for introducing the DPRG approach.

First, the investigation paths spanned all levels of the DPRG (Figure 5.4). Each dot in a graph in Figure 5.4 corresponds to a query or operation performed by a subject. The vertical placement of each dot in the pattern-level represents its concreteness. A dot representing a query for "efficiency" is depicted higher than a dot representing a query for "registration interface". The vertical placement of each dot in the source-level represents its relatedness to the link-level. A query for a source-level entity that was irrelevant to the design pattern was depicted as lower
Figure 5.4: Subjects' Investigation Paths. Gray areas represent actions, dots represent queries. The height of a dot represents the concreteness of the query: a high-dot is an abstract query, a low-dot is a concrete query. Dots shown at in the seed file portion of the link-level indicate link inspections.
than a source-level entity that was relevant to the pattern. The queries and operations are connected to make the progress between levels clearer. These investigation paths show that the subjects chose to browse the levels of the DPRG to make the connections.

Second, at some point, all of the subjects moved sideways in the pattern-level when traversing to, or from, the source-level, rather than merely switching between a point in the pattern and the source-level. For example, the second segment in action A-5 shows a movement from a low-level pattern element, the `synchronize` method of the `MONITOR OBJECT` pattern up to its design context involving serialization and scheduling. This pattern is also visible in B-3, which involved moving up from a more concrete pattern-level node to its design context in the `FORWARDER-RECEIVER` pattern. Similarly, a downward movement is seen in portion C-3 and D-5, in which developers were concretizing their information on `efficiency` in the `OBSERVER` pattern.

The results of the confidence study corroborate both the completeness and lightweightness studies. The results of the completeness study is corroborated by the confidence study since the subjects of the confidence cases were able to investigate all design goals of interest that were described in the design pattern text. The confidence study also corroborates the linking and querying portions of the lightweightness study results. Subjects in the confidence study were able to perform an in-depth investigation into design goals of interest within a reasonable period of time.

5.6 Summary

This chapter has described the evaluation of the claims of the thesis. The thesis makes two main claims: the use of the DPRG approach and tool will enable developers to completely report design goals described in a design pattern and implemented in a code base, and the use of the DPRG approach will increase a developer's con-
idence in terms of how design goals described in a design pattern relate to a code base.

To evaluate completeness, we performed a case study in which we compared the design goals reported by a pair of expert developers to those reported by a DPRG tool user. The patterns used in this study were the THREAD-SPECIFIC STORAGE, CACHING, and STRATEGY. The code base used in this study was the Zen CORBA ORB. This study supported the claim and showed that the DPRG tool user was able to completely report the relevant design goals described in a design pattern.

To evaluate confidence, we performed a case study in which developers from Siemens AG used the DPRG tool to revisit investigation tasks they had previously performed. This study supported the claim, showing that in both cases developers learned design goals related information relevant to their tasks.

This thesis also makes two additional claims: the DPRG tool is lightweight in terms of how much effort is required for its application, and the pattern-level of a DPRG is effective for assisting developers in reporting design detail and context related to entities and concepts that crosscut a design pattern’s description.

To evaluate the lightweightness of the DPRG tool, we performed a pair of case studies in which subjects created a DPRG of the OBSERVER pattern and the JHotDraw code base, and used the DPRG to investigate the concept of efficiency. This study supported the claim of lightweightness, since both subjects were able to create and use the pattern-level of the DPRG in a reasonable time, and one subject was able to fully form the link-level of the DPRG while the other only fell moderately short of this goal.

To evaluate the efficacy of the pattern-level graph we conducted a comparative experiment in which we asked subjects questions about design entities and concepts in the REACTOR and VISITOR patterns whose descriptions crosscut the patterns. The subjects were split into two groups, one given a pattern-level graph, and another given the original design pattern. We noted that those given the pattern-
level graph reported more design detail and context in their responses, and showed more willingness to explore the pattern.
Chapter 6

Discussion

This chapter describes the issues and trade-offs associated with the DPRG approach in general, those that affect its use, and those related the DPRG tool implementation.

6.1 Approach

The DPRG approach is intended to provide a means by which to separately view design goals described in design patterns and to trace those goals to their implementation. We begin by discussing how the searching mechanisms provided by the DPRG tool compare to a common lexical search tool. Then, we discuss extending the DPRG approach to account for overlapping patterns. Then we consider the reusability of pattern-level graphs.

6.1.1 Comparison of the DPRG Approach to Grep

The DPRG approach and tool is not intended to replace either reading or searching through a design pattern. The DPRG approach provides an alternate browsing mechanism for the text comprising a design pattern, and has several differences from a lexical search tool such as Grep [21].

More study is needed to explore how readers respond to graphical query
results versus textual query results, either presented as a separate list of sentences, or as a series of locations within an electronic version of the document. However, certain differences can be identified intuitively.

First, and most obviously, the visualization of the results of regular expression searches offers a collection of information that leads to a different type of exploration than that provided by Grep. The results of a Grep, or regular expression search through a document will lend themselves to a linear investigation of the result sentences or lines of text. The DPRG allows the reader to perform their investigation separately from the intended structure of the document. This affords the reader an abstraction from the text, rather than just a selection from it.

Second, the categorization of nodes helps direct the reader’s investigation of the design pattern. Certain nodes are highlighted as keywords; at a glance, a reader can readily see connections between those keywords.

Finally, the DPRG offers a certain degree of summarization, since it is possible to look at a node corresponding to a design element, and to read the verbs neighbouring it. This enables the reader to visualize how that element relates to other elements in the pattern even before reading on to find out what those elements are. For instance, in the CACHING pattern, by reading only the verb nodes surrounding the Resource User entity, the reader learns the following: the resource user acquires something, uses something, accesses something, and calls something. A developer can read further to resolve the references of these verbs.

6.1.2 Support for Overlapping Patterns

Patterns in code may overlap. A developer wishing to investigate all of the design goals associated with a particular portion of a code base, may not want to perform a separate query for each pattern implemented in the code. Including results from more than one pattern-level DPRG might be helpful. Here, we consider supporting this overlap through overlapping DPRGs.
There are several considerations that should be taken into account before implementing overlap support. First, although there are often similarities in terminology between patterns, the meanings of these terms may differ widely. Because DPRGs present only local context for nodes, it may be difficult or confusing to identify which pattern-level nodes refer to which meaning. Clustering or colouring could be used to differentiate patterns, however, more empirical analysis is warranted before implementation.

Second, it is unclear what benefit, other than the initial convenience, this would provide. Patterns do, at times, refer to one another. It would require semantic analysis of sentences for the DPRG tool to be able to determine, except at a coarse level, which portions of text refer to portions of other patterns. For instance, the Observer pattern names the Mediator pattern in several places. A mediator pattern node could be used in each to represent each of these instances. However, the Observer pattern also notes that a portion of an example, the ChangeManager could act as a Mediator. Would it be necessary for subclasses of the ChangeManager could refer to the relevant portion of the Mediator pattern? Such links would require structural analysis between patterns. Would it be sufficient that the ChangeManager node refer to the Mediator? This would limit the usefulness of representation of the overlapped patterns.

6.1.3 Pattern-Level Reusability

We expect that the pattern-level of a DPRG can be reused for different systems. We have not yet tested this hypothesis. It may be the case that its reusability is limited if the dictionary nouns specified by a developer are specific to a task or code base. We suspect, however, that this is not the case. We have not, as yet, seen evidence that the choices of dictionary nouns are motivated by or affect a subsequent investigation task. For instance, the study organizer formed the DPRGs used in the case studies evaluating confidence and completeness (Sections 5.2, and 5.1) without
in-depth knowledge of the design patterns, or the specific queries that would be performed by the DPRG tool user.

There was no indication, in any of the cases used in validation of the thesis, that the dictionary noun choices should have been specifically tailored to an investigation task. In each of these cases there were instances in which nouns were not included in the dictionary, that could have been included, such as the concept non-blocking in the DPRG provided in the second confidence case study (Section 5.2.3). Had non-blocking been included, the subject in the study could have expanded the non-blocking. Instead, the subject compensated by performing a regular expression search for non-blocking. The dictionary omission did not affect the results of the completeness study.

From the subjects’ comments, we determined that there was never a case in which a dictionary noun node’s inclusion in the dictionary had adverse affects on the subjects’ performance. Even if a frequently occurring noun such as class had been entered into the dictionary, its effect on the usability of the DPRG would have been limited to sentence chain layout alone: all sentences with the word class would have resulted in sentence chains containing the same class node. From these factors we believe that the pattern-level graphs have a high degree of reusability.

6.1.4 Applicability of the DPRG Model to Patterns in General

In considering the DPRG model and general approach, the question arises as to whether it is applicable to all pattern styles. In the validation of the thesis statement, we have applied the DPRG approach and tool to several patterns. These are listed in Table 5.1. These patterns are all described in either the GoF [23] style, or the POSA [11] style. The GoF style generally provides a more compact description of the solution, than the POSA style: GoF patterns are typically near 10 pages in length, whereas POSA patterns are generally over 20 pages. In addition, POSA style patterns often have more intricate descriptions of the solution than the GoF
style patterns. We believe that due to the difference in these two styles, that they are representative of pattern styles in general.

We believe this approach to be applicable to any textually described design pattern. A design pattern, by its very nature, contains a breadth of information that can be usefully displayed in a DPRG.

In on-line design pattern repositories, often a short form of a design pattern will be displayed. These short forms of patterns do not lend themself to formation into a DPRG, since they are, at times, under a page in length. It is likely that when a document is such a size, the usefulness of the DPRG is limited. If a pattern is longer than a POSA pattern, the DPRG approach remains applicable. A long pattern will, of course, lead to a large DPRG, however, since queries are performed to select views of the DPRG, the user of the DPRG tool is rarely aware of the size of the entire DPRG. If a pattern describes a complex solution, as does the REACTOR pattern [61], the views produced by queries will not necessarily be large or complex. The complexity of the views is not dependent on the semantic complexities of a particular design pattern or the solution it describes. Instead, it is dependent on the terms that are used in the pattern description. A query for a frequently occurring term will produce a large view regardless of the size of the pattern, since one sentence chain per instance of the term will be displayed. However, as we can see in Figure 5.3, queries on complex patterns can produce results which are relatively simple. In that figure, the regular expression query "arrive" was performed on the REACTOR pattern DPRG. From this view we can see at a glance that the REACTOR pattern description of the concept of request arrival involves several entities, including the initiation dispatcher, the logging acceptor, and the logging handler.

6.1.5 Attributing Empirical Results to Approach or Tool

The research described in this thesis validated the DPRG tool, which embodies the DPRG model, as a means to relate design goals and their descriptions to relevant
parts of a code base. This validation did not isolate which results were due to the model itself, and which were a result of particular choices in the design of the tool.

Although the validation did not specifically address the validation of the model as opposed to the tool, we believe the empirical results are likely more indicative of the model than of the tool because the tooling is quite common: the node expansion feature is similar to hypertext links, and the regular expression search is similar to any lexical search including grep. On the other hand, the DPRG model was motivated by the need to elucidate concerns that crosscut design documentation, and to provide a collected view of those concerns. This motivation affords browsing of a different style and granularity than that generally afforded by other approaches such as hypertext techniques. These are discussed in more detail in Section 7.3.2. Specifically, the DPRG provides access to information at a term level granularity, rather than at a page or paragraph level granularity, and allows a developer to entirely depart from the original structure of the documentation.

Future research should attempt to attribute empirical results to specific design choices in the model versus the tool.

6.2 Use

This section considers issues related to the use of the DPRG tool. First, we discuss the application of the tool for design pattern exploration and understanding. Next, we discuss the degree to which the DPRG tool tolerates varying user capabilities. Then, we discuss a feature of the tool that allows flexibility in the linking process. Finally, we present limitations in the approach for determining correspondence ratings of links.

6.2.1 Performing Queries Separately at the Pattern-Level

It should be noted that queries at the pattern-level can be performed independently from the link-level and source-level. This means that a developer wishing to explore
the text of a design pattern, but without a corresponding code base, can do so using the pattern-level query mechanisms. We have applied the pattern-level alone in the comparative study in which we evaluated the efficacy of the pattern-level graph (Section 5.4). We found that the use of the pattern-level did enhance developers' ability to collect information about a design pattern. We believe that the pattern-level graph could be used in combination with design pattern text to improve a developers' understanding of a pattern.

6.2.2 Robustness against Varying User Capabilities

The DPRG tool tolerates different degrees of user capabilities in terms of dictionary creation and link-level creation. As noted above, we have found that developers are able to recover from omissions from a dictionary using regular expression queries. Additionally, since the pattern-level can be created quickly, a developer who determines that a phrase was omitted from the dictionary could, with ease, add the phrase and run the DPRG tool to re-create the DPRG.

The DPRG tool can also help a developer recover from some linking errors. In Case 1 (Section 5.2.2) of the confidence study, the subject began by seeding an erroneous link. The inference functionality of the DPRG tool helped to reveal the error. However, in Case 2 (Section 5.3.3) of the lightweightness study we observed an instance of a developer being unable to form the necessary links to complete a task. This shows that while the tool can aid a developer in certain errors, it does not guarantee success.

6.2.3 Code-Fact Alteration to Enhance Linking Flexibility

Code-facts represent the entities and relationships described in the structural diagrams from a design pattern. The relationships between these entities are then compared to relationships between source entities, to obtain a link from the pattern-level to the source-level. At times, an implementation may not use the same relationships
as those set out by the abstract implementation in the design pattern. For instance, in the OBSERVER pattern, the notify method of the subject class calls the update method of an observer, which then alters the value of the ObserverState. This is represented by two code-facts:

\[
\text{notify} \rightarrow \text{update} \quad \text{[label=calls]}
\]
\[
\text{update} \rightarrow \text{ObserverState} \quad \text{[label=writes]}
\]

The basic OBSERVER structure can be altered to implement the push model. In this case, the notify method still calls update, but adds the subject as an argument. Should the developer be aware of the implementation of the push model configuration, code-facts may be added to reflect the structure of the implementation. In this case, the additional code-fact would be:

\[
\text{update} \rightarrow \text{subject} \quad \text{[label=takes]}
\]

Because the developer may wish to use the code-facts for linking in more than one portion of the implementation, we do not allow deletion of existing code facts. In the above situation, the code-facts would correspond to both the push and pull models, and thus would have wider applicability. Additional code-facts may hurt the correspondence rating for a link, since they add edges at the pattern-level which must correspond to edges at the source-level. However, it increases the flexibility of link inference, and improves the chances that useful link possibilities will be suggested by the DPRG tool.

6.2.4 Correspondence Rating Limitations

Inferred links are rated to aid developers by limiting the number of proposed links they must evaluate. Links with a rating below a certain threshold could potentially be ignored.

In validation of the thesis, subjects reported several problems with the correspondence rating features of the DPRG tool. As described in Section 4.3.4 the
rating reports how well the edges of the subgraph neighbouring the source-level node correspond to those in the subgraph neighbouring the pattern-level node of the link pair.

The efficacy of this measure depends heavily on the pattern-level graph. Ratings are more effective when there are edges between nodes neighboring the pattern-level link node. However, such edges are not always present. For instance, as is shown in the link view in Figure 4.9 there is no relationship between the nodes neighbouring the attach method node in the OBSERVER pattern. Simple sub-graphs of this nature are frequent in a source-level graph. The correspondence rating for a link between the attach method and the addToContainer method (also shown in Figure 4.9) would also have obtained a rating of “1”, even though it would not have been accurate.

Developers in our case studies had difficulty intuiting what was meant by the rating, at times regarding a rating of “1” to mean a proof of correspondence, or a low rating to mean that two entities should certainly not be linked. Once the developers were made aware of how ratings are calculated, some opted to ignore them altogether.

Based on these observations, it is clear that other mechanisms for computing correspondence should be developed. It is possible that a solution could be found by examining the behaviour of the subjects who ignored the ratings. These subjects used their own analysis to determine whether a link pair was accurate. As an initial pass, the subjects used lexical analysis to assess the similarity of names, and type analysis to assess whether the linked nodes matched in terms of type, as in method to method links, or class to class links. In cases of doubt, subjects used link inspection to visualize how each node of the link pair fit into their respective contexts. A scheme could be devised where a developer specifies rules for a weighted rating of lexical, type, and surrounding graph structure comparison. More empirical analysis is needed before an implementation choice can be made.
6.3 Implementation

In this section we discuss four issues related to the implementation of the DPRG tool. First, we discuss the use of lexical indicators for link inference. Then we discuss the use of pattern finding technologies to aid in link inference. Finally, we discuss synonym and pronoun resolution in pattern-level graph formation.

6.3.1 Lexical Indicators for Linking

Our current approach for inferring and suggesting links to a developer creating a DPRG does not consider the names of entities at the pattern-level or source-level. In several cases, developers using the tool noted that being able to express certain lexical rules for inclusion, exclusion, or ratings of links would be helpful. For instance, when linking a pattern-level node such as *acquire*, a developer might suggest target strings, such as "get" or "acquire", and suggest exclusion strings such as "set" or "release". The goals for such a feature would be a high-degree of flexibility and appropriate reporting of results to the developer without eliding potentially useful information.

6.3.2 Potential Use of Pattern Finding Technologies

Currently, it is the responsibility of a developer to build the link-level for a DPRG. Although we have provided a lightweight mechanism which facilitates this, we believe that pattern-finding technology could be used to further assist this process. Currently, we are unaware of pattern-finding tools that are sufficiently time-efficient and flexible for this task.

6.3.3 Synonym Resolution

Typically, in design pattern texts, a design entity or concept will be referred to by synonyms. In the *Observer* pattern, for instance, there are synonyms for the
notify method, node, including notify operation, notify method and notify call. Two approaches are available for a developer to include all synonyms related to a concept in a dictionary: The developer can either include all synonyms for the concept or entity, or the developer can edit the text to alter all synonyms to a common name.

There are benefits and shortcomings of each approach. In the first approach, the pattern-level will be formed such that there is a separate node for each synonym of a particular concept or entity. This limits the functionality of node expansion, since expanding a node will not trigger the expansion of the synonym nodes. In the Observer pattern, for instance, expanding the notify method node would not add to the view the sentence chains that include the notify node or the notify call node. Additionally, only one notify node is specified in the code-facts. The node specified is determined by the naming convention used in the structural diagrams in the design pattern. In the case of the Observer pattern, only the notify node is used. Expanding the notify node from the link-level into the pattern-level would only expand sentence chains containing that node. This limitation can be compensated for by the use of regular expression searches. Each of these nodes contains the regular expression notify. Searching for this regular expression will result in the expansion of each of these nodes.

In the second approach, a developer would edit the original text to ensure that all synonyms are replaced with a single term. This has the benefit of collecting all information related to concepts and entities in the pattern-level, and ensuring that all applicable sentence chains are accessible from the link-level. We have found this approach to be onerous and prone to error and omission. For instance, a developer may not notice all of the synonyms used for a particular concept or entity while scanning the design pattern. This manual method degrades when synonyms are missed.

In the validation of this thesis, we used the first approach. This approach had the most potential to harm our results in the study on DPRG completeness
(Section 5.1), however we used it because we did not wish to provide subjects with unrealistic DPRGs in terms of what a developer would likely create. Still, we did not find any instances in which design goals were missed due to the presence of synonym nodes.

An additional approach might be to allow a developer to specify a synonym dictionary. A developer would list noun phrases that should be represented by one dictionary node.

It may also be possible to automatically suggest a synonym dictionary based on lexical similarity amongst noun phrases found in the design pattern text. This approach would need to be validated before its application, since this step may also prove to be onerous for the developer, and may affect the lightweightness of the approach.

6.3.4 Pronoun Resolution

For pronouns to be resolved in a DPRG, the design pattern text must be edited. For instance, in the pair of sentences: “The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are”, the first “it” in the second sentence refers to the subject in the previous sentence. However, this reference will not be represented in the DPRG unless the “It” is changed to “Subject”.

Pronouns in the DPRG are interpreted as new noun nodes, and are thus inserted into the graph without linkage to the concept or entity to which they refer. In this case, there will be a separate node labeled it, rather than a reference to the subject node. Replacing the pronouns with appropriate concrete nouns can provide the proper link. Typically, this editing is only necessary for pronouns at the beginning of sentences. If a pronoun appears in the middle of a sentence, it usually refers to a noun within that sentence, as in the sentence: “In response, each observer will query the subject to synchronize its state with the subject’s state”.

148
Since queries are returned with collections of complete sentences, we are assured that the pronoun ("its") will be attached to the concept or entity to which it refers ("observer").

In the validation of this thesis, we opted not to perform this editing. This choice imposed the risk that subjects in our studies might miss information related to design concepts. It is not possible to use regular expression searches to compensate for pronouns in a pattern-level graph. Developers in the confidence and completeness (Sections 5.2, and 5.1) studies all noted that the presence of pronouns in the pattern-level posed difficulty. However, when we analyzed the results of the completeness study, we found that no design goals were missed by the DPRG tool user. Participants in the lightweightness study (Section 5.3) did not mention being affected by pronouns in the DPRG, and we determined that this was because they were examining portions of a pattern where pronouns were not extensively used.

We chose not to edit the pronouns in the texts of the design patterns in the completeness and confidence studies because we wanted to provide these subjects with the same quality of DPRG as that created by the subjects in the lightweightness study. We decided not require pronoun editing from the subjects in our lightweightness study since it is unlikely that a developer applying the technique independently would choose to perform this task. Additionally, it is conceivable that a text-analysis tool could be applied to perform the resolution of pronouns automatically though our current text-analysis tool, LTCHUNK [47], does not provide this functionality.

6.4 Summary

In this chapter we have discussed various issues related to the approach and functions of the DPRG approach and tool. These issues focused on current limitations of the approach or implementation of the tool, and on ways in which the approach and tool could be enhanced.

Related to the approach in general we considered how the DPRG querying
compared to Grep, a lexical search mechanism. Then we discussed providing support for representing overlapping patterns in a DPRG. Next we discussed pattern-level reusability and posited that pattern-level graphs can be re-used between code bases.

Related to the DPRG tool use, we discussed four issues. First, we considered the use of the pattern-level graph separately to aid in design pattern understanding. Then we commented on the tolerance of the DPRG tool to user capabilities in dictionary creation and link-level creation. Next we discussed the use of code-fact alterations to allow for flexibility in the linking process. Then we discussed limitations for the correspondence rating given to links inferred by the DPRG tool.

Finally, we presented open issues in the implementation of the DPRG tool. First, we discussed the use of lexical information when inferring links. Then, we discussed the use of pattern finding tools and techniques to assist the link inference process. Finally, we discussed synonym and pronoun resolution.
Chapter 7

Related Work

This chapter describes work related to the DPRG approach. Several areas are discussed. First, we consider approaches in which design is encoded at the code level, and then either read by a developer or displayed using specialized tools. Next, we discuss mechanisms which allow analysis between design and code, including techniques for design verification, visualization, and code generation. Then we compare the DPRG approach to research into encoding crosscutting concerns at the design level. Finally, we discuss work related specifically to the pattern-level of the DPRG, including concept graphing and abstraction, and assistance with design pattern understanding.

7.1 Encode Design Concepts in Code

In this section we consider approaches proposed for encoding design information at the code level. Three such approaches are considered. First, we discuss information transparency, then literate programming, and finally design pattern comment information.

Developers can look for design intent in the code itself through information transparency. In information transparency, programmers encode design information into their code by means such as naming and coding conventions. Concepts are
seeded into the code one by one, and are best integrated at the time of program creation, which limits such a technique in terms of legacy system use. Mechanisms, such as [27], can then be used to search through the code base and expose portions pertaining to a particular concept. While retrieving the concepts can be lightweight, information transparency relies heavily on the willingness of the original designer to encode as many concepts as possible. Also, there is a risk that the conventions used to expose certain concepts will be violated when the system undergoes change. In the DPRG approach, because the linking of concept to code is based on entities and relationships in the code base, adapting a DPRG to a changed system means re-building the source-level and updating the link-level. Since the DPRG approach is designed to work with existing systems, it more automatically deals with changes to the underlying source code.

A widely known technique in which high-level information is integrated into the code itself is known as literate programming [38]. In addition to the original, WEB [38], many literate programming tools are available. An example is CWEB [39], a pre-processor that supports literate programming in C and C++. In CWEB, programmers use mark-up language features to distinguish between different sections in their program. Each section contains a text part and a code part, or chunk. The code parts can be extracted so that the program can be compiled. The entire program, or web, can be typeset for easier reading. As with information transparency, literate programming presents a bottom-up approach. Because the description portions are separate from the code itself, it would be straightforward to document an existing code base, however this would require significant knowledge, and should be left up to someone who understands the code. The DPRG approach is intended to aid developers who do not have intimate knowledge of the code they are examining, while requiring nothing of the original developers. In this sense, applying the DPRG approach to existing code is less involved. However, literate programming can be applied to any code base, whereas the DPRG is limited
to systems that apply design patterns.

PCL is a design pattern specific approach for encoding design information at the implementation level. In a study on the usefulness of design pattern information in program maintenance, Prechelt et al. provided a group of subjects an implementation of design patterns augmented with pattern comment lines (PCL) [57]. It was established that the use of PCL helped developers in the assigned program maintenance task. PCL includes only structural information related to design patterns. No rationale behind the implementation is described. Though useful, PCL poses the same problems as information transparency, since the comment lines are inserted manually, and by a knowledgeable developer. The DPRG approach differs from this approach in that it is intended for use by novice developers with no necessary participation on the part of the original developer.

### 7.2 Analysis Between Design and Code

There has been a significant amount of research into how developers can relate design to code and code to design. Here, we discuss three categories of approaches: design verification, extrapolation of design information, and code generated from design.

#### 7.2.1 Verify Design

Investigating conformance to a particular design can be performed in an implementation specific way, or with respect to more general structure. Here we discuss representative examples of both techniques. As an example of checking for implementation specific posited structure, we discuss the RM tool [51]. This approach is both flexible in terms of the kinds of structures that can be addressed, and lightweight.

Software Reflexion Models (RM tool) [51] allow the developer to check the conformance of system code to intended structural properties. This mechanism is intended to help programmers understand high-level structural design information
as opposed to functional design information.

We use PatternLint [64] as an example of conformance checking against a general structure. PatternLint is intended to facilitate checking whether a pattern is implemented correctly. Code is analyzed for information like calls and variable dependence between classes, and stored in a fact base. Facts describing structural features of patterns are also introduced. A developer may then compare the system against a pattern description using a series of conformance and non-conformance rules. The DPRG is a less formal approach than this for linking pattern design information to source. For PatternLint to locate a pattern, all the conformance and non-conformance rules must be satisfied. Partial checking is not provided, whereas with the DPRG approach, if portions of a source system relate to portions of a pattern, they can still be linked to the pattern description. Also, the use of Prolog makes PatternLint a heavier-weight technique in terms of preparation for use.

We see structural checking techniques as complementary to the DPRG approach. While the linking functionality of the DPRG tool is intended to assist developers understand how design goals relate to a code base, the structural rigour provided by these tools provides a different kind of analysis and support than that provided by the DPRG tool. It is conceivable that the DPRG approach could be used in combination with such approaches. Once a developer has determined that a particular design has been used in their code base, the DPRG tool could be applied to assist in understanding the rationale behind the implementation.

7.2.2 Extrapolating Design

To extrapolate design information from code, several kinds of approaches have been proposed. First we consider pattern mining, in which design pattern information is extracted from a code base, next we discuss program visualization, in which visual abstractions are formed from code base information, and finally we consider automatic clustering, which involves categorizing portions of code based on conceptual
Pattern Mining

Pattern mining techniques help a developer search for and recognize design patterns used in the source code comprising a system. For instance, the Pat [56] system presented by Prechelt and colleagues uses Prolog and a commercial CASE tool to locate instances of structural design patterns in source code. The SPOOL [34] system combines various source code capturing tools, including SniFF [9] and Gen++ [12], with pattern detection mechanisms to form a database that can be queried to report on structural features of the code base. The program visualization tool Program Explorer tool presented by Nakamura et al. [42] uses a Prolog fact base that contains both static and dynamic information to help filter and visualize design patterns found in the code.

These approaches have three major differences with the DPRG approach. First, the linking portion of the DPRG tool is intended to help a developer locate portions of code related to a particular pattern. The intent of the linking portion of the DPRG tool is not to provide a way of searching for instances of a design pattern in a code base. It is our belief that such a feature would be a helpful addition to the DPRG tool, and we intend to explore how pattern finding approaches can be integrated to that end.

Second, the DPRG linking process is intended to be lightweight, and to be very tolerant of large variants, or partial implementation of patterns in the code. Most pattern finding tools are constrained in terms of what kinds of patterns they can locate, and also their flexibility in terms of design choices for the pattern implementation. We have yet to find a pattern finding tool that can locate instances of more complex POSA [11, 61] patterns, for instance.

Third, pattern finding tools provide a link between the structural elements of a pattern, and the implementation of those elements. The information presented can
be compared to that provided in the link level of the DPRG. The DPRG however, allows browsing into the relevant portions of text for the structural portions of the pattern linked. In this way, the developer is able to see not just that a particular pattern implementation exists, but why relevant design choices were made.

Program Visualization

Developers can also extract structural and behavioural information from source without making use of mining techniques. Program visualization [3, 58] involves providing the developer with a visual abstraction of certain features of source code based on the source itself. While some such tools extract and provide visualizations of performance related information [74], and some look at interaction patterns to help elucidate system behaviour [32], those most similar in motivation to the DPRG approach, such as [59, 31] provide graphical models to help the developer understand design intent. Richner et al. [59] provide a mechanism for customized generation of views of the source code extracted from static and dynamic information about the source. Developers can query using set operations on programmatic constructs such as methods and classes, and in terms of relationships such as contains and invokes. We see this and similar techniques as complementary to the DPRG approach. These mechanisms provide views of code that do not link to documentation. In terms of structural and behavioural analysis of code, these techniques are preferable to the DPRG. The DPRG tool provides minimal functionality for browsing at the source-level.

Automatic Clustering

In automatic clustering, rather than providing mechanisms for querying existing software structure or behaviour, tools offer restructurings of the source code based upon the optimal grouping of similar features. In general, an assignment of code portions to clusters is based on call and variable dependence relationships. In some
tools, such as BUNCH [48], the clustering is assigned based on minimal call and variable dependence relationships between clusters, and maximal relationships within. In others, such as the Automatic Query Language (AQL) [60], clustering is assigned based on the best match to a target structure set by the developer. The model is described in terms of relationships such as "file includes library", "function calls function" and "function uses identifier". In general in automatic clustering, results are reported both in terms of module-to-module relationship views of the proposed structure, and also code level views. The DPRG approach differs from these mechanisms in that it is not intended to directly help with a re-modularization of source code. By the same token, these techniques are not intended to support linkage to high-level design intent.

7.2.3 Generate Code from Design

Generative programming represents the effort to generate software artifacts based on abstractions such as crosscutting concerns, or design patterns. The applications related to design patterns are intended to help developers who either have trouble translating a design pattern into an appropriate implementation for their needs, or those who find such a task onerous [10]. Here, we discuss two techniques.

Budinsky, Finnie, Yu and Vlissides propose a tool to assist a developer in applying a design pattern [10]. They augment the pattern description itself with hypertext features to exploit the cross-referencing between portions of the pattern description. Some links are at the page-level, allowing developers to browse the sections of a design pattern, and browse between design pattern descriptions. Other links are embedded within the text of the page, and point to other sections of the design pattern. This approach also offers searching of the design pattern text. In addition to the sections of a design pattern itself, a "code generation" page is provided. This code generation page allows a developer to specify the names of the participants in the design pattern, and which trade-offs should be made in the code
generation. Goals such as notification efficiency may be available for the Observer pattern. This approach is closely related to the DPRG approach and tool. The aim of this approach, however, slightly different from that of the DPRG approach. Since this is intended to help in the application of design patterns, rather than understanding code that has been created using design patterns, the emphasis of facilitation is different, and complementary to that of the DPRG approach. This approach does not provide assistance for determining which tradeoffs have been made in existing applications of design patterns. The DPRG approach, however, has been shown to help in elucidating which design choices were made at the time of implementation (Section 5.2). The hypertext approach does provide navigability and searchability of the design pattern text. However, as we saw in the experiment on the efficacy of the pattern-level graph (Section 5.4), searching is not always sufficient to help a developer understand a design pattern, or determine the design context of concepts and participants in the pattern. This technique bears similar limitations to other hypermedia mechanisms discussed in Section 7.3.2, in that it constrains the granularity of the target of links to pages of information rather than individual sentences, as is provided in the DPRG approach.

Florijn, Meijers and van Winsen offer tool support for the implementation, integration and conformance checking of design patterns in Smalltalk [24]. Their Fragments approach splits design patterns into elements such as classes, methods, and objects, and connects them through roles, or slots. Roles are constrained so as to limit which type of fragment can fulfill a particular role. Conformance checking of design patterns is applied by validating the type of fragment that fills a particular role in the implementation. Fragments are generated to address unfilled roles. Fragments does not allow a mechanism with which to expose or check for crosscutting design goals at the implementation level.
7.3 Follow Links from Design to Code

To trace design concepts down to code, two main approaches are proposed. Knowledge based approaches encode abstract domain information and provide querying capabilities. Design rationale tracing techniques propose mechanisms for capturing historical information related to implementation, or ways in which to explore existing artifacts related to design rationale.

7.3.1 Knowledge Bases

Knowledge based techniques provide the developer with the ability to search for higher-level concepts involved in a software system. The techniques that are closest in motivation to the DPRG are LaSSIE [17], where the knowledge base is built before the tool is used, and DESIRE [8], where knowledge is incrementally included in a domain model as developers use the tool. Both of these techniques have the limitation that it takes significant up-front time to build up the knowledge bases before concepts can be queried. In contrast, the DPRG tool obtains design goal and rationale information from existing documentation.

LaSSIE attempts to combat design invisibility and complexity by allowing developers to query about actions and actors in the source code. For instance, in examining a private branch exchange phone system, a developer could ask “what actions by a bus controller are caused by an attendant”. Such querying ability has been shown useful to developers trying to understand the design of a large software system, however, LaSSIE has some drawbacks. First, because the knowledge base only contains actors and actions, there is no way to encode context information such as “why is this action performed here” or “is this operation involved in more than one feature”. The DPRG approach has no such inherent limitation. However, whether such information will be present for a given portion of code depends on the presence of the information in the pattern text. Second, the knowledge base of actors and actions is built manually, which makes start-up time for use of the
tool an issue. One of the key features of the DPRG is that it takes relatively little
time to set up and begin to use. This short ramp-up time offsets the fact that the
information linked will not account for all the functionality of the code base.

DESIRE allows developers to query a code base for "human level concepts". For example, in a debugging system, a developer could ask to see all the portions
of code related to the "set breakpoint command". In DESIRE, identification of a
concept involves examining the typical features that characterize the concept, its
relationship with other concepts in the domain, relevant domain knowledge (such as
synonyms or nicknames for terms) and the syntactic or conceptual context likely to
occur. Concepts are stored in and retrieved from a domain model. The knowledge
in the domain model is built up incrementally as developers use the tool. This
means that for conceptual information to be readily available to a developer, it
must have been recorded in the past. If it has not been, then the developer can use
the code analysis facilities available (a program slicer, lexical searching, structural
visualization) to help build up the concept anew. If a developer is not interested
in taking the time to build up a concept, and the concept does not already exist
in the domain model, then DESIRE is of little help. With the DPRG, developers
can browse from any high-level information to follow it through the design and, if
matched to source, then also get pointers to relevant portions of code. Granted, if
the concept is not included in the design pattern that was the basis for the DPRG
then that conceptual information is not available. However, if the concept of interest
is described in the pattern, then it is available "for free".

7.3.2 Design Rationale Tracing

Research into tracing design rationale can be split into the capture and collection of
design rationale artifacts, and the use of existing artifacts to provide design rationale
information. We discuss techniques from both of these research efforts.
Capture and Collection

Many Design Rationale approaches are focused on collecting design rationale information. IBIS [15], for instance, is a method for structuring and representing content of design discussions. IBIS provides an electronic forum in which discussions are textually depicted and stored. The IBIS process generally starts with a root question, such as "how to improve customer satisfaction with a particular product". Subsequent questions are asked, and the issue is subdivided until all issues and sub-issues are addressed. All paths of discussion are recorded.

Peña-Mora and Vadhavkar [54] proposed the Design Recommendation Intent Model (DRIM), which is a mechanism for capture and propagation of design rationale related information. The DRIM system stores practitioner experience, including reports of successful implementation, and areas of difficulty while applying design patterns. This stored experience is used to augment the design patterns with design rationale information and create a corporate memory of the application of design patterns. This information is then used as a basis for making intelligent choices as to whether to apply design patterns in other situations.

We see the DPRG as complementary to design rationale collection approaches. While these techniques are generally focused on recording the history of design decisions for a particular implementation, the DPRG approach focuses on allowing deconstruction of existing artifacts which refer to a general design solution.

Existing Artifacts

Other approaches allow exploration of existing artifacts. Two approaches closely related to the DPRG approach are hypertext approaches, and text analysis approaches.

Hypertext approaches allow a non-linear navigation of documentation. Some hypertext systems, such as CHIME [18] allow linking within source code to allow browsing from variable definitions to uses, and from a caller to a callee. Some, such
as SODOS [30] offer linkage between documents without linkage to source. The most
related to the DPRG approach are those techniques, such as SLEUTH [22], that link
design documentation to source. In SLEUTH, links within the documents are gen-
erated when the author specifies filters, indicated by regular expressions. The links
can point to files, or to anchors within documents. The within-document anchors
must be inserted manually. There are several main differences between SLEUTH
and the DPRG technique. One of the main assumptions of SLEUTH is that doc­
umentation will be designed for hyper-linkage. In contrast, the DPRG technique
has no such limitation. Another difference is the granularity of linkage to code. In
SLEUTH, links to code are at the file level. This limits the degree of precision with
which documents in SLEUTH can describe portions of the code. This is an inher­
ent problem in SLEUTH, because it operates directly on source files, into which
hypertext anchors are inserted. In the DPRG there is no inherent limitation to the
granularity of links. This allows developers to query on very specific portions of the
source code, and, where possible, retrieve appropriate design context. In addition
to limitations on granularity, usability issues arise in hypermedia documents. In
a study performed on a hypermedia system which allowed a developer to compare
and contrast concepts in a document [52], it was noted that use of the hyperme­
dia features were more harmful than helpful to reader comprehension. This can be
combined with a sense of being ‘lost in hyperspace’ [16] in which readers are unable
to trace or mentally model the path of their hypermedia investigation.

Antoniol et al. used text analysis to form links from documentation to source
code [1]. They applied a language model in which probabilities are assigned to every
string of words taken from a prescribed vocabulary. In their approach, text and code
are analyzed. The design documentation text is used to estimate a language model.
The code is analyzed and translated into an intermediate and interpretable form.
They then assign probabilities that indicate whether a portion of code resembles a
document. They applied the technique in order to associate manual pages to the
source code they describe. We see this approach as complementary to the DPRG approach. This approach works at the granularity of document level links to code, whereas the DPRG approach links from within a document. Using this approach, a developer could narrow down which documents relate to which portions of code. The developer could then use the DPRG approach (if it were extended to general documentation) to perform a finer grained analysis of the links.

7.4 Express Crosscutting Concerns in Design

Since the introduction of language support for the advanced separation of concerns [71], efforts have been made to allow expression of crosscutting concerns in design.

Several approaches have been proposed. Among the most recent, are Clarke et al.'s Theme/UML [14], Stein et al.'s UML-based aspect-oriented design notation [68], and UMLAUT [29]. These approaches all provide conventions or mechanisms with which to apply the Unified Modeling Language (UML) to design for languages that support crosscutting concerns. Theme/UML, in particular, suggests a new UML construct called a composition pattern (CP). To design a system, a pattern is used to define each participant in the design, and each concern that crosscuts the participants. In the Observer pattern, for instance, two base pattern classes are defined, one for the subject, and one for the observer. The subject is defined as an object whose state changes, and the observer is defined as an object that relies on the subject’s state. An additional pattern defines the shared element of the state change itself. These three patterns are then bound together using binding rules. In this way participants and the concerns which crosscut them are associated.

The approaches in this area are complementary to ours. These techniques do provide a variety of ways to map crosscutting concerns at the design level. However, they do not provide means with which to record the reasoning behind the design decisions. For instance, in the Observer pattern example above, the reasoning
behind the design choices is not encoded in the composition patterns.

7.5 Pattern-Level Related Work

This section discusses three areas of work related to the pattern-level graph alone. First, we discuss conceptual graphs, which the pattern-level structure resembles. Next, we discuss proposed mechanisms for abstracting and summarizing concepts from text. Finally, we discuss other approaches for assistance in understanding and applying design patterns.

7.5.1 Concept Graphing

The pattern-level graph is similar to the structure used in Conceptual Graphs (CGs) [67]. CGs are visual systems of logic that are readable by humans. Similar to the pattern-level of a DPRG, CGs represent concepts and the relationships between concepts. Conceptual Graphs have been used for many purposes, including the checking of consistency between multiple views of a software specification [67]. Expressing design patterns as CGs could be beneficial as this expression would enable formal analysis of the pattern. However, since patterns are written in free-form text, the text would have to be massaged heavily before such a representation would be possible. Given our intent to use the pattern level graph to visualize the relationships between entities in the pattern rather than analyze the pattern, the extra effort required to mould the pattern text into a CG is not yet warranted.

Sunetnanta and Finkelstein make use of conceptual graphs in their technique for automating consistency checking in multiple-perspective software specifications. They noted that each participant in a software development process has a different view of the specifications that contribute to its implementation. The heterogeneity of these perspectives make it difficult to perform consistency checking and integration on the specifications themselves. They suggested the use of conceptual graphs as a common medium for describing specifications, and then exploited the formal
nature of the conceptual graph to check for consistency and aid with integration. This approach is similar to the DPRG approach in that it intends to allow for a representation of multiple design goals through a graphical notation, in this case, the conceptual graph. However, the DPRG approach differs in that the pattern-level graph is intended to separate design goals whereas the ViewPoints framework used by Sunetnanta and Finkelstein is intended to integrate and check consistency of design goals. The DPRG approach is not intended for formal analysis of this kind.

Related to concept graphs are goal graphs, as introduced by Chung et al [13]. In this case, graphs are manually constructed to organize and account for non-functional requirements for a system. Goals are nodes, each representing a requirement, which describes both the name of the goal, and also the concept which should be considered with relation to it, such as “security”. Goals are decomposed into offspring goals, according to a set of guidelines. This recording and refinement is done within a framework, which requires the capture of expert knowledge about the relevant non-functional requirements. The goal graph approach and the DPRG approach are aimed at different parts of the software lifecycle in which different information is available: goal graphs are intended for use early in the software development process when developer expertise is available, whereas the DPRG approach is intended to assist developers in evolving or understanding an existing system through analysis of documentation that describes it.

7.5.2 Concept Abstraction

Many approaches exist for re-structuring text according to topics [28, 50]. A well-known technique is the TextTiling algorithm, proposed by Hearst [28], which partitions documents into units consisting of one or more paragraphs based on topic. Portions of text are deemed a portion of the same topic based on cosine similarity analysis, in which takes into account both the words themselves and their placement
within an area. These techniques differ from the pattern-level graph approach in that they re-structure text rather than allowing a developer to traverse the text regardless of its structure. Additionally, they do not allow for the extraction of minor themes in the text. Since an individual design goal may be mentioned only as an aside within the discussion of another topic, it is unlikely that these techniques would re-structure text so that these goals would become more explicit.

7.5.3 Design Pattern Understanding

Several efforts have been undertaken to clarify the meaning and presentation of patterns using formal representations. For instance, Lauder and Kent [43] present a three-model approach that involves a role model, the most abstract and "pure" representation of the pattern, a type model, which refines the role model, and the class model, which forms the concrete implementation. LePUS [19] is a notation based on conventional logic for representing design patterns. It enables reasoning about both the structure and meaning of design patterns. Mikkonen applied the DisCo [49] specification method based on the temporal logic of actions as a means of helping to improve the rigour of pattern-oriented development.

All of these approaches can be used to help clarify potentially ambiguous parts of a pattern. They can also be used to help reason about pattern integration. DPRGs are complementary to these approaches in that they can help a developer understand an existing pattern sufficiently to formalize the pattern. However, in addition, DPRGs can help a developer understand why parts of the pattern exist: the formalization techniques do not include this why information.

Pattern Hatching [72], introduced by Vlissides, is focused on teaching the application of patterns, and helping developers write patterns themselves. The goal of the book complementary to our own, in that it aims to provide a reasonable point of view on when and how to apply design patterns as described in the GoF pattern text [23]. Pattern Hatching helps developers strike a balance between applying a
pattern according to good design, and taking the original pattern prescription too literally.

7.6 Summary

In this chapter we have compared the DPRG approach and tool to several areas of related work. We identified five areas of research that related to our work.

The first area includes techniques that allow a developer to encode design concepts at the code level. These techniques differ from the DPRG approach in that they require prior knowledge of design concepts of interest, and might be limited in terms of the kinds of rationale which can be encoded.

The second area includes approaches for analysis of the design in terms of the code. We began with design verification tools, which check the structure of a code base against a particular set of characteristics. These approaches are complementary to the DPRG approach, since they do not allow for browsing and deconstruction of the document level, nor do they intend to provide explicit links to design goals and rationale. Then we reported on mechanisms which extrapolate design from code. Such approaches include pattern finding techniques, which closely relate to the linking capabilities of the DPRG approach and tool. Like the design extrapolation approaches, these do not provide links to goals and rationale, and in addition are often not lightweight to apply. Then we discussed generative programming approaches, which differ in intent from the DPRG approach in that they are meant to assist developers creating software while the DPRG approach is aimed at assisting developers in understanding existing code bases.

The third area involves research into linking from design to code. In this area, we first examined knowledge based approaches, which differ from the DPRG approach in that they involve the creation of conceptual information about a code base as opposed to relying on existing design rationale documentation. In addition, these techniques are non-trivial in terms of application effort. Next, we discussed
design rationale tracing approaches. Some research in this area is focused on capturing design rationale information, hence is complementary to the DPRG approach. Other research in this area focuses on using existing documentation to link design to code. These approaches differ from the DPRG approach in that they generally provide inter-document links. The DPRG approach allows decomposition of a single document, linked to portions of code. Some of these techniques also require effort by the authors of either the documentation or the code. The DPRG approach can be applied without involvement of the original code or pattern authors.

The fourth area identified is the expression of crosscutting concerns in design. This area of work is complementary to the DPRG in that it provides new ways in which to describe design concepts that do not align with an object-oriented structure. They do not provide a mechanism for encoding the rationale associated with the design choices.

Finally, we identified work related to the pattern-level of the DPRG. We discussed concept graphing, which provides a more formal means by which to encode the rationale of a document. Such approaches might be difficult or inappropriate to apply for pattern-level formation since they require significant analysis of the documentation and focus on formalization. Next, we discussed concept abstraction techniques that decompose text into portions related to particular topics. These techniques differ from the DPRG in that they do not allow a developer to browse a document to trace concepts through it. Then we discussed approaches for assisting developers in understanding design patterns. In contrast to the DPRG approach, these are often not applied with relation to a particular code base, and are not intended to allow a developer to freely explore the design goals and rationale described in the design text.
Chapter 8

Conclusions

'I don’t think they play at all fairly,' Alice began, in rather a complaining tone, 'and they all quarrel so dreadfully one can’t hear oneself speak—and they don’t seem to have any rules in particular; at least, if there are, nobody attends to them—and you’ve no idea how confusing it is all the things being alive; for instance, there’s the arch I’ve got to go through next walking about at the other end of the ground—and I should have croqueted the Queen’s hedgehog just now, only it ran away when it saw mine coming!'

—Alice In Wonderland, Lewis Carrol

It is not always clear why code is the way it is. We know that when software developers do not know the design rationale behind code, they may make changes which violate goals for the structure and behaviour of a system.

The motivation for this dissertation is to ease the task of tracing how and where design goals are carried out in a system. Specifically, this research has focused on developing a tool to help a developer explore design rationale set out in design patterns.

The thesis of this research has been that an approach that allows a developer to navigate through design documentation and to explore how design goals are
carried out in a code base can help a developer more completely identify relevant
design goals, and feel more confident about how the design goals relate to the code
base.

To provide such an approach we have developed the Design Pattern Rationale
Graph approach and associated tool. A DPRG is a graphical representation of the
text of a design pattern, linked to portions of a particular code base. A developer
uses the DPRG tool to form a DPRG, and to select views of the DPRG. The DPRG
approach is lightweight, in that the approach can be applied with limited effort, and
within the context of one investigation task.

We demonstrated the validity of these claims through several case studies,
which varied in size and style. To address developers’ confidence with respect to
design goals, we performed a study in which two developers from Siemens AG re­
peated an investigation task they had performed in the course of their work. These
studies showed that the developers learned information related to design goals in
their code bases. To address the claim of completeness of information related to de­
sign goals, we conducted a case study in which we compared design goals specified
by experts and those found by a DPRG tool user. The results of this study showed
that developers can find complete design goal information by using the DPRG tool.
We also performed studies to address the additional claims of lightweightness and
pattern-level efficacy. The lightweightness study examined two cases in which sub­
jects were given a specific task. The pattern-level efficacy study was comparative,
and examined the accuracy of participants’ responses to questions about design
patterns.

### 8.1 Contributions

In addition to the development of the DPRG tool and the demonstration of the
validity of the thesis statement, this research makes four contributions.

First, the pattern-level graph may aid developer understanding of a design
pattern. In our studies, we have learned that developers have difficulty gleaning design detail and context from a textual description of a pattern. We also found that the use of the pattern-level of a DPRG eased this task.

Second, the pattern-level formation approach itself indicates that a useful representation of a textual document can be obtained through minimal syntactic and lexical analysis of a parts-of-speech tagged text. Obtaining a parts-of-speech tagged text is straightforward, as is the syntactic analysis needed to form the sentence and sequence chains. The graphs are connected lexically, in that dictionary nouns are points of connection between sentence chains. This is an inexpensive approach that does not require significant computation time or resources.

Third, the research provides evidence of the usefulness of partial linkages between design patterns and source. In our case studies it was typically the case that the link-level formed by the subjects only linked portions of the pattern to source. This partial linking enabled the subjects to explore design goals in a partially implemented pattern. For example, the DPRG tool user in the completeness study (section 5.1) identified design goals that were relevant to the portions of the patterns implemented in the Zen [76] code base, and listed no irrelevant design goals.

Fourth, this research shows the importance of flexibility in linking a design pattern to a code base. In all of the example systems we examined in our validation of this thesis, no pattern implementation conformed precisely with the abstract implementation provided in the original design pattern. The nature of the linking features of the DPRG tool afforded tolerance for mismatching of types and design options between the pattern and the implementation. The subjects were able to obtain useful linkages despite mismatches.

8.2 Future Work

There are many remaining avenues of exploration for how a lightweight link from design to source could benefit a developer. Several such avenues are discussed in this
section. These include application of the DPRG approach and tool for evaluation and comparison of design patterns, and the extension of the DPRG approach to design documentation and design rationale artifacts.

8.2.1 Evaluating Design Patterns

Because a DPRG provides a means with which to decompose a design pattern to separate out design goals, it is conceivable that a design pattern author could benefit from applying a DPRG in evaluating how well a design pattern describes design concepts. Design pattern writers are often writing with a particular implementation or set of implementations in mind. An author could use the DPRG to link a design pattern description to a code base while writing the text. The author could apply the pattern-level to various implementations of the pattern to compare how well the pattern descriptions correspond to the known uses of the pattern.

8.2.2 Comparing Design Patterns

Design patterns often have many structural similarities. For instance, the class diagrams of the COMPOSITE pattern and the DECORATOR pattern have large portions which look similar. However, the meaning behind the structure and the design goals which the patterns implement are different. In the course of writing and evaluating patterns, the pattern community often compares the solutions described in a new design pattern to existing solutions. Structural comparisons are inadequate for this task, and another means of comparing patterns may be useful. The DPRG approach may be applicable to this task, by forming a DPRG of each pattern, and identifying design goals related to the participants of each pattern. This would support comparison of participants of proposed patterns in terms of the design goals relevant to them.
8.2.3 Extension to Design Documentation

Although DPRGs were designed and intended for use with design patterns, we believe this approach could be extended for use with design documentation created as part of the software lifecycle. For design documentation to be usable with the DPRG approach, it would have to contain information spanning from the source code structure of the implementation to the rationale for design decisions that lead to the implementation. It may be possible to merge high-level design and detailed design documentation to obtain this information. Avenues of exploration might include studying how such an approach would affect developers' interest in both creating and referring to design documentation specific to a particular system.

8.2.4 Lightweight Linking of Design Rationale Documents

Our research has shown that lexical means, such as that used to connect information to dictionary nodes, can be useful when exploring design goals in a design pattern. One of the problems faced by the Design Rationale community was maintaining a traceable link between artifacts describing the requirements and goals for a system. We believe that lightweight ways of linking these artifacts, such as the lexical model used in the DPRG, may provide some such links without the developer effort commonly required.

8.2.5 Use During Software Development

The DPRG approach is intended for use in software maintenance. However, in one of our case studies (Section 5.2.2), we observed that developers implementing a design pattern may need assistance in determining how their implementation relates to the design pattern description. The DPRG approach may be suitable for this task. A developer who is implementing the design pattern could build the link-level during development, and then use the DPRG to track which design goals need to be addressed by which portions of the implementation.
Bibliography

documentation links in OO systems. In Proceedings of the Working Conference

1996.


patterns with design rationale graphs. Technical Report UBC:T2-2002-01, Uni­
versity of British Columbia, Canada, 2002.

crosscutting concerns during software evolution tasks: An inquisitive study.
In Proceedings of the International Conference on Aspect-Oriented Software

programs. In ACM Conference on Object-Oriented Programming, Systems,
Languages, and Applications: Workshop on Specification and Design for Object-


[8] T.J. Biggerstaff, B.G. Mitbander, and D. Webster. Program understanding and
the concept assignment problem. Communications of the ACM, 37(5):72–83,
May 1994.


[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


176


automatic clustering to produce high-level system organizations of source code. 
In IEEE Proceedings of the International Workshop on Program Understanding, 


[50] N.E. Miller, P.C. Wong, M. Brewster, and H. Foote. TOPIC ISLANDS - A 
wavelet-based text visualization system. In Proceedings of the IEEE Conference 

[51] G. Murphy and D. Notkin. Software reflexion models: Bridging the gap be­
tween source and high-level models. In Proceedings of the Symposium on the 

[52] D.S. Niederhauser, R.E. Reynolds, D.J. Salmen, and P. Skolmoski. The in­
fluence of cognitive load on learning from hypertext. Journal of Educational 


[54] F. Peña-Mora and M. Vadhavkar. Augmenting design patterns with design 
rationale. Artificial Intelligence for Engineering Design, Analysis and Manu­

[55] N. Pennington. Stimulus structures and mental representations in expert com­
prehension of computer programs. International Journal of Computer and In­

[56] L. Prechelt and C. Kramer. Functionality versus practicality: Employing ex­
isting tools for recovering structural design patterns. Journal of Universal 

[57] L. Prechelt, B. Unger, M. Philippsen, and W. Tichy. Two controlled exper­
iments assessing the usefulness of design pattern documentation in program


179


Appendix A

Design Patterns Primer

Design patterns were first introduced in the field of architecture by Christopher Alexander. Alexander described design patterns as existing in two contexts: in the world, and as elements of language. In the world, design patterns represent a relationship between a problem and a solution. As elements of language, patterns are instructions for solving problems as they arise in the world.

Alexander’s intention was that a set of design patterns could be used as a language, called a pattern language with which to express design needs of a client. Alexander’s vision was that through the use of pattern languages, inhabitants could design their own environment according to their own needs.

Design patterns were intended for the design of cities, including descriptions of the use of ring roads and parallel roads; buildings, with prescriptions for positive outdoor space; areas for couples and suggestions for privacy within buildings; and construction, outlining column placement and choices of materials.

Alexander provided a form with which to define a design pattern. First, the context, or situation, is described. To refine the description of a context, examples are given. The reader of a pattern can determine whether a pattern is appropriate by seeing whether their situation fits into the described context. Once the context is established, desirable outcomes are described, and certain solutions, or design
forms, are given.

The *bed cluster* pattern, for instance, described a context in which there is a need to configure sleeping arrangements for a group of children. Three options are given: Two extreme options, in which the children are either bunked together, or kept apart, and the recommended compromise, in which children sleep in alcoves with a common place to play. The *bed alcove* pattern can be used in the implementation of a *bed cluster*.

Software engineering researchers began to realize the importance of the fact that, as in cities, buildings, and physical construction, recurring patterns exist in software design. Design patterns were adopted into computer science as a means to both record solutions to common design problems, and also to provide a language with which to express requirements and ideas.

The first intentional application of Alexander-style design patterns in software engineering occurred in 1987, when Cunningham and Beck provided a five-pattern language to empower a group of clients to design their own user interface for an application. Patterns in the language included specifications for windows and menus [6].

Since this implementation, research in the area of software design patterns has become more refined. Several patterns books have been published describing different kinds of patterns, including the pattern style presented by the so-called *Gang of Four* [23], and the Patterns of Software Architecture [11, 61] style.

A description of “anti-patterns” has also been published. Anti-patterns describe the outcome of using software design solutions that seem attractive, but that have negative consequences. Examples of anti-patterns are *Design for the Sake of Design*, in which the aesthetics of the design are focused upon to the detriment of its function, and *Big Ball of Mud*, which describes a mass of code which is tangled and disorganized.

While pattern styles differ, they retain the original intent of patterns as set
out by Alexander. As in architecture, patterns are intended for use together, as a means to describe entire systems.

All software design patterns contain certain essential components [23]: a name, which provides a conceptual abstraction for the design; a problem statement, which describes the objectives for a particular context; the context itself, which describes the scope of applicability of a design pattern; the forces, which are a description of both the motivation for a solution, and trade-offs to consider; the solution, which describes the structure and behaviour of the design; examples, which illustrate a simple case of the solution; the consequences of the pattern, indicating possible outcomes of its application; the rationale which justifies the design of the pattern; related patterns, to place the solution in context with other solutions; and known uses of the solution.

As an example, we present these components for the OBSERVER pattern.

- **Name:** OBSERVER

- **Problem:** Many objects depend upon the state of another object.

- **Context:** This pattern should be applied when a change to one object requires changes to another object; when an abstraction has two aspects which need to be used independently, but which should be aware of each other's state; when an object should update others as to changes, without tightly coupling between them.

- **Forces:** Avoidance of tight-coupling between the objects; Efficiency of updates.

- **Structure:** The OBSERVER pattern involves two classes, a subject, and an observer. The observer registers to receive notification of updates from the subject. The subject may notify the observers in different ways, and may track the observers in different ways.
• **Example:** The example given in pattern description is of charting programs which all rely on the state of a spreadsheet. The spreadsheet plays the role of the subject, the chart programs (which create graphs of the data on the spreadsheet) play the role of the observers. Another example is of an analog and digital clock that always keep time with each other. Class diagrams, sequence diagrams, and sample code is shown which provide a framework for applying the pattern.

• **Consequences:** There are two positive outcomes from the use of this solution. First, the subject and observer are abstractly coupled. Second, the broadcast style communication enhances the flexibility of the system's design. One potential negative outcome is that performance may suffer if updates are too frequent.

• **Rationale:** The rationale for the use of the OBSERVER pattern is spread throughout the pattern description. One example is the reasoning behind the push model notification convention, which can be applied to enhance efficiency.

• **Related Patterns:** Two patterns are listed as being related to the OBSERVER pattern: the MEDIATOR and the SINGLETON patterns. A MEDIATOR acts as a go-between for a pair objects. An OBSERVER could act as a MEDIATOR if one object wished to observe the other object's changes. An OBSERVER could be a SINGLETON (an object with just one instance) to become globally accessible.

• **Known Uses:** Several uses of an observer are given. One in a Smalltalk environment and another in a user interface toolkit.

Individual design patterns and pattern languages are being proposed almost constantly. The pattern authoring culture has been largely responsible for the quality and applicability of the designs described. The series of Pattern Language of Programs (PLoP) conferences place strong emphasis on the process of writing a
pattern. Authors are paired with domain experts, known as “shepherds”, who work closely with the authors to hone the design description before it is presented at the conference. Patterns are then presented in a forum, called a writing workshop, in which strict feedback conventions are followed. Research into ways to compare and evaluate patterns is actively being pursued.