Design Pattern Rationale Graphs: Linking Design to Source

Elisa L.A. Baniassad & Gail C. Murphy
University of British Columbia
2366 Main Mall Vancouver Canada V6T 1Z4
{bani,murphy}@cs.ubc.ca

Christa Schwanninger
Siemens AG, CTSE 2
Otto-Hahn-Ring 6, 81739, Munich Germany
christa.schwanninger@siemens.com

Abstract

A developer attempting to evolve a system in which design patterns have been applied can benefit from knowing which code implements which design pattern. For instance, the developer may be able to understand the purpose, or to assess the flexibility of the code, more quickly. The degree to which the developer benefits depends upon their understanding of the pattern. Achieving an in-depth understanding of even a simple pattern can be difficult as pattern descriptions span several pages of text, and discuss interrelated design concepts and choices. To enable a developer to effectively trace the design goals associated with a pattern to and from source, we have developed the Design Pattern Rationale Graph (DPRG) approach and associated tool. A DPRG makes explicit the relationships between design concepts in a design pattern, provides a graphical representation of the design pattern text, and supports the linking of those concepts to implementing code. In this paper, we introduce the DPRG approach and tool, and present case studies to show that a DPRG can, at low-cost, help a developer identify design goals in a pattern, and can improve a developer's confidence about how those goals are realized in a code base.

1. Introduction

> A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. ... [A design pattern] describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use. [6, p. 3-4]

A developer who has chosen to implement a particular design pattern as part of building a software system must make choices amongst the implementation alternatives presented in the pattern. For instance, in the OBSERVER design pattern [6], which provides a solution for notifying registered dependents (observers) when a particular object (a subject) changes state, the developer must decide how to transmit information related to the change. One option is the push model in which the subject sends all observers detailed information whether or not each observer needs it: This approach can limit the reusability of observers. Another option is the pull model in which the subject may send little information, requiring interested observers to request needed details: This approach can be inefficient because an interested observer has to try to determine what changed.

A developer who is later asked to perform an evolution task involving this code, such as adding a new feature or fixing a defect, can benefit from knowing that it represents the OBSERVER pattern. For instance, the developer may be able to understand the purpose of the code more quickly, to posit how the code works, or to understand the flexibility of the code. The developer may be able to recognize the pattern by comments in the code, by the names chosen for the classes or methods in the code, through the use of a pattern finding tool (e.g., [18]), or through the use of a pattern generation or encoding technique (e.g., [22, 8])

The degree to which a developer benefits from knowing that a piece of code implements a particular pattern depends upon the developer's understanding of the design pattern. If the developer knows many of the possible implementation variants and understands their relationship to design goals, the developer can work with the code effectively. Achieving an in-depth knowledge of even a single pattern can be difficult as the pattern description may span several pages of text and may include several diagrams. The OBSERVER pattern, for instance, is eleven pages long and discusses more than seven significant implementation choices, including how to store the subject-to-observer mapping, and how unexpected updates can be handled. Trying to understand which variant was implemented, why that variant was chosen, and the effect of that variant on design goals requires careful reading and analysis of the pattern. In the open source and industrial code that we have examined, the original developers have seldom provided this kind of *rationale* in any comments or

documentation associated with the code. A developer encountering the code must thus recall or find the rationale in the pattern. In a small exploratory study we conducted involving the Visitor [6] and Reactor [19] design patterns, we found that pattern-experienced developers had difficulty answering detailed questions about how the pattern worked based on a reading of the pattern alone [1].

To enable a developer to trace the design goals associated with an implementation of a pattern to and from source code implementing the pattern, we have developed the Design Pattern Rationale Graph (DPRG) approach and associated tool. A DPRG makes explicit the relationships between design concepts in a design pattern, provides a graphical representation of the design pattern text, and supports the linking of the design concepts to the implementing source code. With a DPRG, a developer can navigate from a design concept in the pattern to other relevant parts of the pattern, and to the implementation of the concept in source code. A developer can also navigate from a class or method in the source to relevant parts of the pattern. This support helps a developer reason about their code in the context of the relevant design concepts. Our tool is lightweight: A DPRG can be built and linked to source at reasonable cost.

In this paper, we present the DPRG approach and tool in the context of a simple scenario (Section 2). We also present a number of case studies we have conducted with pattern-experienced developers to validate the utility of the approach (Section 3). In the latter part of the paper, we discuss outstanding issues with our approach (Section 4), compare to related efforts (Section 5), and summarize (Section 6).

2. Design Pattern Rationale Graphs

To introduce the concept of a DPRG, and to explain how a DPRG can be used by a developer, we consider a simple scenario. In this scenario, a developer has been asked to improve the efficiency of a drawing editor built on the JHotDraw object-oriented framework [9]. The design of JHotDraw relies on several design patterns, including OBSERVER. In our scenario, the developer has decided to investigate the efficiency of a part of the code implementing the OBSERVER pattern. We describe the format of the DPRG, how one is created, and how one is used.

2.1. Overview

A DPRG consists of three levels (Figure 1): pattern, source, and link. Our current DPRG tool requires one DPRG per instance of a design pattern in the source.

The pattern level consists primarily of sentence chains—graphical representations of the sentences comprising the text of the design pattern of interest. The left area of

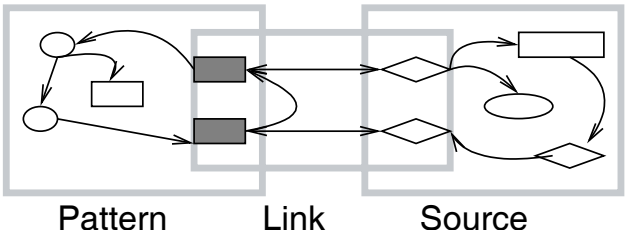

Figure 1. Three Levels of a DPRG.

Figure 2 shows two sentence chains from the OBSERVER DPRG including one corresponding to the part of the following sentence from the pattern "pull model may be inefficient because Observer classes must ascertain what changed without help from the Subject" [6, p.298]. Noun phrases in a sentence chain are are shown as rectangular nodes; verb phrases are shown as oval nodes.[1] Edges associating noun and verb phrases are annotated with the parts of the sentence that serve to connect the phrases in the pattern text. A developer reads this sentence from the DPRG by locating the root of a verb phrase chain, such as `may be`. The subject of this verb can be found by following edges into the verb, in this case, `pull model`. The object of the verb can be found by following edges out of the verb, such as `inefficient`. Diamond-shaped nodes are sequence nodes; when order matters, they indicate the order in which sentence chains should be read.

Noun phrases that represent a pattern design element are represented by a single (gray rectangular) node in the DPRG pattern level, enabling the DPRG to bring together all parts of the pattern text related to that element.[2] For example, the `observer` node in the DPRG in Figure 2 is a design element node, and thus brings together, in this fragment, two sentences related to the concept from the pattern text. Dashed edges associate these nodes with phrases that modify them. In this figure, there are two such associations: *update efficiency*, and *observer classes*. When reading sentences, such node-pairs should be read together.

The source level contains information about structural relationships derived from the code base. Nodes in this level are entities from the code, such as classes and methods. Edges are relationships between entities, representing structural associations such as calls between methods, or inheritance relationships between classes. In the area marked *Source Level* of Figure 2, seven source level nodes representing JHotDraw elements are shown. A developer can determine from this representation that the `PolyLineFigure` class extends the `AbstractFigure` class, that `AbstractFigure` is contained in the `standard` package and

[1] A noun or verb phrase may consist of a single noun or a single verb.

[2] Our tool colour codes the nodes according to their role in a sentence to make it easier to read a DPRG.

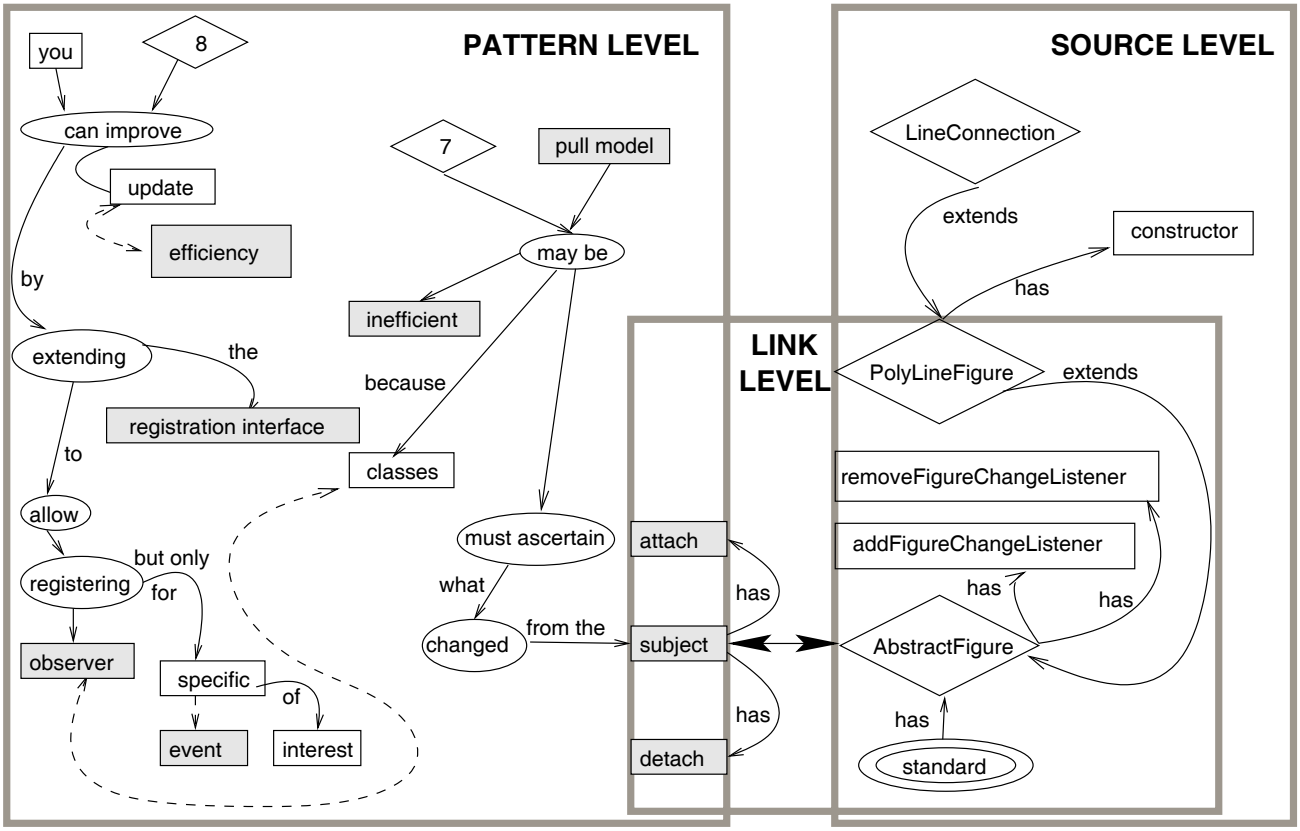

Figure 2. Three queries on a DPRG of OBSERVER/JHotDraw: a regular expression query for **efficien, a link level expansion of *subject*, and a source level expansion of `PolyLineFigure`**

has two methods, `removeFigureChangeListener` and `addFigureChangeListener`, and that the `LineConnection` class extends the `PolyLineFigure` class, which has a constructor.

The link level represents the association between elements from the design pattern and entities in the source. The link level contains nodes from the pattern level, nodes from the source level, and edges associating the two. The left most portion of the graph in Figure 2 shows a portion of the link level where the pattern-level `subject` node has been associated with the JHotDraw `AbstractFigure` class.

2.2. Creating a DPRG

A developer must create each level of the DPRG. Effort invested in creating the pattern level can likely be amortized over the linking of that pattern level to different implementations of the pattern. Effort invested in creating the source level can be amortized over the linking of that source base to different patterns.

Pattern Level Creation Creating the pattern level requires three inputs: the text comprising the pattern, a dictionary of design elements—participants, and concepts—specific to the pattern, and an encoding of the structure section of the pattern.

Providing the text is straightforward: it can be extracted from a digital representation of the pattern. The only sections that are not used from the pattern text are those that relate to sample code, and to known uses 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 onerous; it took one of the authors less than 10 minutes to annotate the text of the OBSERVER pattern.

To help the developer provide the dictionary of keywords, our DPRG tool extracts 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. For the OBSERVER DPRG, the design elements identified included `change request`, `notify` and `ob-`

server. Nouns not chosen as design elements included need, state, class, and call. It took developers in one of our studies (see Section 3), approximately 10 minutes to identify 28 (of 400) noun phrases from the OBSERVER pattern text as design elements. A developer using a DPRG can easily refine the design elements, and regenerate the DPRG pattern level, if needed.

Providing the encoding of the structure section consists of extracting the participants and their relationships from the structural diagram within the pattern, and recording the information in a simple textual format. The relationships of interest are the same as those described below for the source level. These associations are shown in Figure 2 as link level edges between the pattern level nodes: The subject has attach and detach methods. This information could be automatically analyzed and extracted from a digital representation of the pattern; our DPRG tool does not yet have this capability.

Given these inputs, the DPRG tool automatically creates the pattern level. Using a parts-of-speech tagger called LTCHUNK [15], the DPRG tool identifies the noun and verb phrases in each sentence of the pattern text. Each sentence is then processed individually. Except for noun phrases included in the dictionary, each occurrence of a noun or verb phrase introduces a new node into the pattern level. 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 causes the introduction of an edge between the source and destination nodes. 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.

Source Level Creation To create the source level, the developer runs a third-party program database tool that can extract entities and relationships—source models—from the code base. For the JHotDraw example, we used the Chava tool [13], which operates on Java class files, and post-processed its output with a simple script to massage the data into the format expected by our DPRG tool.

The source models we used for the examples and studies presented in this paper included: the *extends* relation between classes, the *calls* relation between methods, the *has* relation between any source entity, such as a class has methods, the *accesses/writes* relations describing when a method reads/writes the value of a field, the *implements* relation between classes and interfaces, and the *takes* relation where a method takes a type as a parameter.

Link Level Creation Links between the pattern and source levels in a DPRG are specified by the developer through the use of the DPRG tool's *links view*. To initialize this view, the developer provides one or more seeds: A seed describes an entity in the source level that the developer believes corresponds to an entity from the structure section of the pattern. For instance, for the OBSERVER example, the developer may seed an entry associating the FigureChangeListener interface from JHotDraw with the observer pattern entity. Positing this relationship requires minimal perusal of the code, and some understanding of standard Java coding conventions.

Given this seed, the tool can infer additional links. For instance, after two iterations, our tool presented a number of possibilities in the *links view*,[3] including the following:

```
U, update
U, subject state
N, subject, AbstractFigure
A, attach, addFigureChangeListener
A, detach, removeFigureChangeListener
A, notify, figureInvalidated
```

Each entry in the links view depicts its status: a U indicates an unlinked pattern-level node entry, such as subject state; an N indicates a link inferred by the tool, such as subject-AbstractFigure; and an A indicates a link a developer has accepted as accurate, such as attach-addFigureChangeListener. From the *links view*, a developer can inspect a link, exclude unwanted links, or infer additional links.

Inspecting a link results in a display of a portion of the DPRG that shows the pair of nodes comprising the link, and all nodes which have an edge to or from one of the nodes in the pair. Edges between these surrounding nodes are also included. Inspecting the inferred subject-AbstractFigure link results in display of the link level shown in Figure 2. The context shown in this view supports the inferred link since the methods of AbstractFigure could play the roles of the attach and detach methods from the pattern.

Excluding a link flags the link in the view as excluded. An excluded link is not considered in further inference. At any time, a developer can reaccept a previously excluded link.

When a developer chooses to infer additional links, the tool iterates across all seeded and accepted links. For each such link, the tool compares the relationships emanating from the source entity node with those emanating from the pattern level node. Where the relationships agree, the end points of the matching relationships are suggested as new links.

A developer continues seeding, inferring, and excluding links until satisfied that the pattern is linked to the code base. In one of our studies (Section 3), developers were able to link JHotDraw to OBSERVER, and to investigate the connection in the context of an assigned task, in approximately one hour despite being unfamiliar with the code base.

[3]The *links view* is a simple textual window. We have chosen not to reproduce the window for space considerations, showing only the content.

2.3. Investigating a DPRG

To select relevant portions of the DPRG to view to aid with the task at hand, a developer can use two kinds of query operations: regular expression searching, and node expansion.

A regular expression query applies to the pattern level, and results in the inclusion of all sentence chains containing nodes that satisfy the expression, and all sequence nodes that point into those sentence chains. This kind of query is used to focus on a particular concept within a pattern. The developer can begin investigating efficiency in an implementation of the OBSERVER pattern using this kind of query by searching for the regular expression **efficien** to capture nouns involved with both efficiency and inefficiency, such as *efficient*, *efficiency*, *inefficient*, and *inefficiency*. The result of this query is shown in the pattern level of Figure 2. Viewing these results, the developer sees that the "pull model may be inefficient".

Node expansion queries support navigation between different levels of a DPRG. For example, expanding the *subject* node in the link level of Figure 2 reveals a node representing the `AbstractFigure` class in JHotDraw, which corresponds to the `subject`, and also reveals the other link level nodes neighboring the *subject* and the `AbstractFigure` nodes. To better understand the context of `AbstractFigure`, the developer can expand the class `polyLineFigure` at the source level, resulting in the source level portion of Figure 2, in which all source level nodes neighboring the expanded node are displayed.

To further investigate *efficiency*, the developer decides to expand the `pull model` pattern level node for more information about the concept (results not shown). Expanding a node at the pattern level results in a view of all sentence chains and all sequences that contain the expanded node. This query results in a DPRG that includes the following fragments of sentence chains: "the pull model sends minimal notification", and the pull model "emphasizes the ignorance of the subject". One sentence chain mentions another option, the *push model*, as an alternative strategy.

Based on an investigation of the results of these queries, the developer determines that efficiency might be improved by investigating the push model. Changing to the push model would likely require modifications to the notification mechanism. Thus, the developer decides to perform a regular expression query for **notif**. The sentence chains displayed include a node called `notify method`. To see how this node relates to the code base of interest, the developer expands the `notify method` node at the link level, and finds the `FigureInvalidated` method. The developer now has a concrete location in the code from which to consider the change task.

3. Case Studies

We make three claims about the DPRG approach:

1. Confidence — Access to a DPRG will improve a developer's confidence about which design goals are relevant to a (part of a) code base.
2. Completeness — Given a DPRG, a developer can effectively navigate to all design goals related to (a part of a) code base.
3. Lightweightness — Developers can create DPRG at reasonable cost.

We use the term design goal to refer to a property desired in the system, such as avoiding race conditions, or easing the introduction of a new algorithm.

We chose to evaluate each of these claims in a separate case study.

3.1. Confidence Study

The purpose of this study was to investigate whether access to a DPRG can improve a developer's confidence that design goals are achieved in a code base. The study consisted of two cases. In each case, the subject, a developer from Siemens AG, was asked to use a DPRG to revisit an investigation task they had recently finished. The investigator provided the pattern level of a DPRG for a pattern chosen as relevant by the subject, and a source model of the subject's code base. The subjects were given a 35 minute tutorial on how to use a DPRG. The subjects were alloted one hour to apply the DPRG tool to their system. During a session, the subjects worked only with the DPRG.

Validity We informed each subject that the intent of the study was to observe their use of the DPRG tool when investigating design goals. When we analyzed their use of the tool and their responses to an interview, we focused on what information a subject learned about their system. If the developer learned something about the goals associated with their code, we consider their confidence was increased. We did not ask subjects directly about their confidence to avoid hypothesis guessing and evaluation apprehension. The subjects did ask questions about the DPRG approach during a session; the investigator was careful to limit responses to the use of the DPRG and did not provide direct assistance with the task. Our choice of using tasks completed previously by the subjects may have affected the results because the subjects may have been unwittingly wary of admitting they missed information when they performed the task.

Case #1 As part of an earlier task on a system that supported the visualization of object connections in a distributed system, the subject had implemented a locking mechanism to allow only one method of an object to execute

at one time. The subject later learned about the MONITOR OBJECT pattern [3], which synchronizes concurrent method execution within an object to ensure only one method runs at any given time within the object. The subject believed that this design pattern closely represented her implementation. The subject had attempted to match her implementation to the pattern previously with limited success.

Actions The subject started by seeding three links, including an association between a class, `Monitor Condition`, in the pattern and a method, `TransmitCallID`, in the code base. The subject then used the DPRG tool to infer additional links, resulting in a suggestion by the tool that the `TransmitCallID` method be associated with the `synchronized method` pattern element. After inspecting the suggested link, the subject wanted to know more about synchronization in the pattern and thus chose to search for "block" at the pattern level. This query resulted in 18 sentences, one of which included the phrase: "if a synchronized method must block or cannot make immediate progress, it can wait on one of its monitor conditions". The subject then revisited the inspection view of the `synchronized method`–`TransmitCallID` link, and expanded the `synchronized method` node at the pattern level. From the resulting graph, the subject learned about the relationships between the `synchronized method` and the `monitor object`.[4] Specifically, 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 their initial seeding of the `TransmitCallID` method with the `monitor condition` class was incorrect, and that the proposed link from the `synchronized method` to the `TransmitCallID` method was correct. The subject then expanded the `monitor object` at the pattern level, revealing the caller of the `synchronized method` to be the `client`. After some investigation, the subject noted that the methods exported by the `monitor object` are often synchronized.

At the end of the session, the subject stated that they believed the code matched the MONITOR OBJECT pattern.

Results Before using the DPRG, the subject was unsure whether her code implemented the design pattern despite having spent time attempting to make the correspondence. The subject also did not have a good understanding of the pattern as can be seen from her initial seeding attempt. Before accepting the tool's suggestion on a different link, the subject investigated the context for various portions of the pattern. After the study, 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 pattern design goals of method synchronization, and of allowing only one synchronized method in an object to run at a time to prevent race conditions. The subject was also able to articulate that the code and the pattern shared the drawback that it may be difficult to change the synchronization policy. We infer that this subject's confidence about their understanding of the relationship between design goals and the code base increased after using the DPRG.

When asked whether the DPRG tool had been helpful, the subject said:

> 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.

Case #2 The second subject was instrumenting methods in a system pertaining to certain design patterns to enable the generation of 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, and reported being familiar with the code base.

Actions The subject began by positing one seed that related a class in the code to 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, as possible links for the `sendCmdAsynch` and `sendCmdSynch` methods in the code. The subject did not recognize the role of the `marshal` method. To investigate, she inspected the suggested link, and 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.

The subject then chose to investigate the `sendCmdSynch` and `sendCmdAsynch` methods. She was reminded that the code base to which she was adding tracing had taken two approaches to implementing the pattern. Given the linking of each of these methods to `marshall`, she was unsure of the pattern's intent with relation to asynchronous versus synchronous approaches. To investigate further, she performed a search for "asynch", which resulted in several sentences, including one stating that the developer "must de-

[4]The `monitor object` is used by the `monitor condition` class.

cide whether the receiver should block until the message arrives". The subject said that she was unware that the pattern had not required both the asynchronous and synchronous options, but she now understood that the original developers had implemented both options.

The subject then performed a pattern-level search for the regular expression "blocking", which resulted in several sentences, including the following: "if the underlying IPC mechanism does not support non-blocking, the developer could use a separate thread to handle communication". The results of the query also included the information that certain timeout values were involved in the non-blocking implementation. The subject indicated that this was likely the approach taken in the asynchronous version of the implemented code.

Results Using the DPRG tool, the subject learned 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 from the pattern level to investigate the differences until she was satisfied the differences were consistent with intent of the pattern. The subject also learned 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 able to then explain the code. We infer from her explanations that her confidence about which design goals were met in the system increased after the use of the DPRG tool.

3.2. Completeness Study

To verify that relevant design goals could be identified from portions of code, we conducted a case study that involved two industrial software developers with expertise in both patterns and a particular code base, and one DPRG tool user (an author on this paper) who was unfamiliar with both the patterns and the code base. The experts worked together to report design goals relevant to certain portions of the Zen CORBA ORB [23]. The implementation of this system was based on a number of design patterns. Based on information provided by the experts, the investigator created several DPRG's. The tool user was then asked to report the design goals related to the code linked in the provided DPRGs. The user used expansion queries from the link level up to the pattern level to prepare the results reported. Three design patterns were used in the study: CACHING [11], THREAD SPECIFIC STORAGE [3] and STRATEGY [6].

Validity To ensure the DPRG tool user did not fish for high-level goals, the user was not informed of any target design goals, and was restricted in the kinds of searches allowed.

Results The tool user identified six of the seven goals identified by the experts, and noted eleven additional goals. The one goal missed was the *avoidance of blocking overhead* in the THREAD SPECIFIC STORAGE pattern. This goal may have been missed for several reasons. First, *blocking* is not mentioned specifically in the pattern, although *locking* is mentioned extensively. Thus, the goal could not be directly derived from the DPRG. Second, none of the pattern level entities related to locking were linked in the DPRG. Since the user was working from linked elements up, the goals related to locking were not encountered. When the expert participants were asked to examine the list of eleven additional goals, they reported that all were considered applicable, though of secondary importance. From these results we conclude that the DPRG representation and tool do enable a developer to effectively navigate to the relevant high-level design goals.

3.3. Lightweightness Study

To be useful, it must be possible for a developer to create and use a DPRG at reasonable cost. To investigate whether our approach meets this goal, we conducted a study in which two industrial developers from Siemens AG were given the task of finding portions of JHotDraw code related to efficiency in the OBSERVER pattern using a DPRG that they had to create. We provided the subjects with the source level for JHotDraw, and the massaged text and encoded structure section of the OBSERVER pattern. We provided these inputs as the need to provide this information is dependent upon the current state of the tool, and is not central to the approach. To create the DPRG, the subjects had to create the dictionary to be used in pattern level creation, and had to create the link level.

Validity The developers had access only to the DPRG tool. The task we chose stressed information across the levels of the DPRG. The session time allotted was sufficient for the task.

Actions Each participant was able to create the dictionary, and the initial unlinked DPRG with relative ease; the first subject completed in 10 minutes, the second in 15 minutes. The subjects then began the task of finding the code related to efficiency. Completing this task required linking the pattern to the source level. The first subject completed the task in 70 of the 120 allotted minutes, reporting that the code implementing the subject's registration interface effects efficiency. The second subject, after the full 120 minutes, had correctly identified several links including an `observer` and a `subject`, but was unable to locate the attach method.

Results The first subject demonstrated that it was possible to create a DPRG for an unfamiliar code base and complete the task in a short amount of time. The second subject was able to address efficiency at the pattern level, but had taken a source-centric approach to linking, and was hindered by the inability of the DPRG tool to support browsing

of the source model and source. This subject spent considerable time reading the source model itself in database form.

3.4. Synthesizing 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 3). Each dot in a graph in Figure 3 corresponds to a query or operation performed by a subject. The queries and operations are connected to make the progress between levels clearer. These investigation paths show that the subjects did not intuit how design goals likely played out in the source, nor how the code related to the goals. Instead, they chose to browse the levels of the DPRG to make the connections.

Second, at some point, all of the subjects moved through 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. These actions are shown in grey in Figure 3. For example, the second grey portion of (A) 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) in the first grey segment, 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) and (D), in which developers were concretizing their information on *efficiency* in the OBSERVER pattern.

Although the generalization of our results is limited, we believe our claims about the DPRG approach likely apply to other development settings for three reasons. First, the confidence study used two realistic cases; industrial developers examining their own systems and revisiting investigation tasks that they had previously performed. Second, across the three studies, six patterns from four sources were used. Finally, the results of the less realistic completeness and lightweightness studies are corroborated by the experiences of the developers participating in the confidence study.

4. Discussion

Why not use grep? A DPRG of a pattern, even when not linked to source, can be used to understand a pattern. However, the DPRG approach, whether linked or not linked to source, is not intended to replace either the reading or textual searching of design patterns. Instead, it provides complementary support. Whereas a lexical search typically results in a user sequentially visiting each result in the text, the results of a regular expression search on a DPRG are a graph that presents the information separately from its structure in the document. An obvious limitation of this presentation is a lack of context for a sentence matching a query. On the other hand, a different presentation of the sentence may lead to a different focus on the information by a user. The ability of a DPRG to collect crosscutting information and present it in the context of a design element can also make it easier to understand some aspects of a pattern. For instance, in the CACHING pattern, it is possible, by reading on the verb nodes surrounding the `Resource User` element, to learn that the resource user acquires something, uses something, accesses something, and calls something.

Overlapping patterns With our current tool, a DPRG can represent only one pattern. Since one portion of a code base may represent more than one pattern, it may be useful to consider extending a DPRG to represent multiple patterns whose implementations may or may not overlap. There are two hurdles to providing this support.

First, the same term may mean different things in different patterns. Since a DPRG presents only local context for nodes, it may be difficult to identify which pattern-level nodes refer to which pattern and which meaning. Visual cues or filtering to differentiate the sentence chains from different patterns might be sufficient to overcome this hurdle.

Second, the developer gains from a DPRG representing multiple patterns when it is useful to display information in response to a query from multiple patterns. It is an open question whether it is helpful to show the results if they are not integrated. An integrated display, for instance, would make explicit the relationships between the patterns, as when one pattern refers to another. Any significant integration of pattern information would require a deeper automatic semantic analysis.

Improving link inference Our current approach for inferring links does not consider the names of pattern or source model entities. Several developers who have used the tool have noted that an ability to express lexical rules for inclusion, exclusion, or prioritizing of links would be helpful. For instance, when linking a pattern level method with a name like “acquire”, a developer might suggest target strings, such as “get” or “acquire”, and might suggest exclusion strings such as “set”, or “release”.

Pattern level reusability We expect that the pattern level of a DPRG can be used many times for different systems. We have not yet tested this hypothesis. It may be that its reusability is limited if the dictionary choices made by a developer are task- or source-base specific.

5. Related Work

Expressing Design in Code With literate programming, Knuth suggested that programs should be human-readable

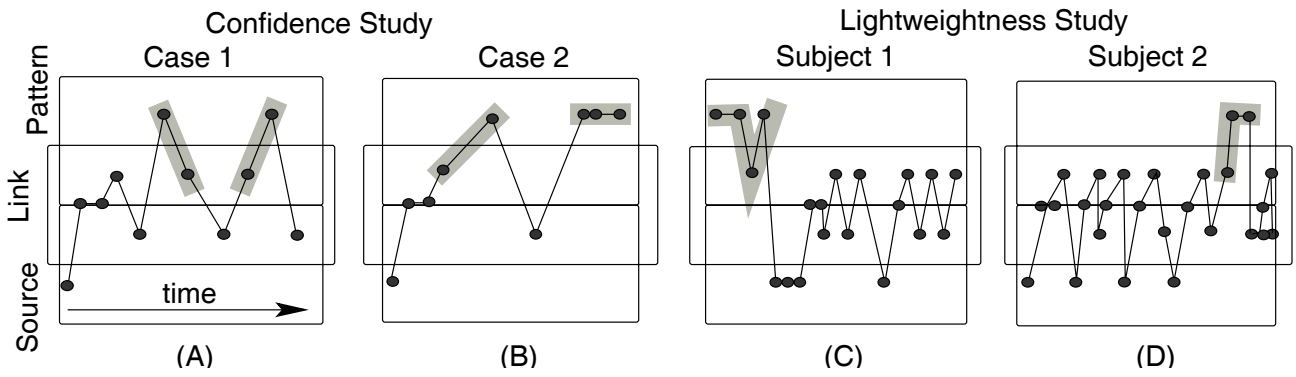

Figure 3. Subjects' Investigation Paths

in addition to being executable [12]. In this style of programming, documentation and code are combined in a single artifact: tools are then used to produce appropriate views for humans or computers. Griswold has advocated the embedding of knowledge about interdependencies between parts of a system into code through the use of information transparency techniques, such as such as naming conventions, formatting style, and placement within the source [7]. The degree to which a developer evolving the code benefits from these kinds of approaches depends on the quality and extent of the design expressed in the code. Although DPRGs apply in a narrower context, they can provide deep design information for pattern-related code even when such information has not been encoded beforehand.

Several researches have investigated generative and language-based approaches to ease the use, and subsequent recognition, of design patterns in code. For instance, in the Generic Pattern Implementation [22] approach based on C++ templates, a developer tags a location in the code where a particular pattern is to be used; code to configure the concrete instance of that pattern can then be generated automatically. As another example, Hannemann and Kiczales have shown how aspect-oriented programming can improve the modular expression of some kinds of patterns, improving the reusability, composability, and pluggability of patterns [8]. These approaches can help a developer use and recognize patterns, but they do not help a developer understand the subtleties of a pattern and its implementation

Inferring Design from Code Reverse engineering techniques produce abstract models of a software system, such as a subsystem diagram (e.g., Rigi [16]) or a sequence diagram (e.g., Shimba [21]), from source code or execution traces. These models can help a developer understand how a system works, but they do not describe *why* particular choices were made in the implementation. Pattern mining techniques help a developer search for and recognize design patterns in the source code of a system (e.g., [18, 10]). DPRGs are complementary to pattern mining techniques: A DPRG can be used to help a developer understand the pattern once found.

Verifying Design Design conformance tools check whether the implementation of a system conforms to an intended design. PatternLint is one example tailored to design patterns [20]. Rules about conformance and nonconformance to the structure of a pattern are specified in a logic fact base. These rules are compared against structural features, such as the calls between classes, extracted from source code. PatternLint can help a developer understand how an implementation may or may not structurally correspond to a pattern. The DPRG approach can also help in assessing conformance; it is less automatic than PatternLint but it can provide more detail about implementation variants, and can be used to match against partial implementations of a pattern.

Capturing Design Rationale A number of techniques help a developer capture design rationale and link it to code. These techniques vary in what rationale is captured, how it is captured, and how it is represented. In the LaSSIE system [4], for instance, an action-object format is used to capture and integrate architectural, conceptual, and implementation information about a system. In the DESIRE framework [2], an object-oriented domain model is used as the basis to guide the recovery of design concepts, and to link those concepts to a code base. These techniques both require a significant investment of developers time to encode the knowledge base prior to its use. The DPRG approach can be applied at much lower cost, albeit to a narrower range of code.

Other techniques focus on lower-cost capture and linking. Lougher and Rodden tool, for instance, allows developers performing maintenance tasks to annotate code with text and graphics, enabling the capture of rationale and the sharing of information over time [14]. The SLEUTH tool takes a generative approach by helping a developer to link existing design documentation to source [5]. Links are formed based on regular expressions specified by a devel-

oper. Links within the design documents are limited to pre-specified anchor points; links to source are limited to files. In contrast, the DPRG approach links design information on the basis of dictionary words that can be easily re-specified, and supports links to specific program elements. Both Lougher and Rodden's tool and SLEUTH require the maintenance of information added to development artifacts, whereas the DPRG approach can be applied on demand.

6. Summary

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 [17]. When ignorant surgery occurs, the structure of a system tends to degrade, leading to increased costs for subsequent evolutionary tasks.

Design Pattern Rationale Graphs (DPRG) are intended to reduce occurrences of ignorant surgery by making the design rationale behind parts of a system associated with design patterns accessible to a developer. A DPRG makes the relationships amongst design concepts in a pattern explicit, and enables the linking of the concepts to the parts of a system implementing the pattern. The cost of creating and linking a DPRG is reasonable; a developer can create a DPRG within the context of a particular evolutionary task.

Through a series of case studies, we have shown that DPRGs show promise in improving a developer's confidence about the design goals related to a piece of code, that a developer can effectively access all relevant design goals represented in a DPRG from code, and that the approach can be applied within a short amount of time.

7. Acknowledgments

This work has been funded in part by Siemens AG and in part by a University of British Columbia fellowship. We would like to thank all of the participants in the case studies for their time and feedback. We thank M. Robillard, J. Hannemann and Y. Coady for insightful comments on an earlier draft.

References

[1] E. Baniassad. *Linking Design to Source Using Design Pattern Rationale Graphs*. PhD thesis, Univ. of British Columbia, 2002. To appear.

[2] T. Biggerstaff, B. Mitbander, and D. Webster. Program understanding and the concept assignment problem. In *CACM*, pages 72–83, 1994.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, volume 1. Wiley & Sons, 1996.

[4] P. T. Devanbu, R. Brachman, P. Selfridge, and B. Ballard. Lassie: a knowledge-based software information system. In *Proc. of Int'l Conf. on SE*, pages 249–261, 1990.

[5] J. C. French, J. C. Knight, and A. L. Powell. Applying hypertext structures to software documentation. *Info. Proc. and Mngmt*, 33(2):219–231, 1997.

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

[7] W. Griswold. Coping with crosscutting software changes using information transparency. In *Proc. of Reflection 2001*, pages 250–265, 2001.

[8] J. Hannemann and G. Kiczales. Design pattern implementation in Java and Aspectj. In *Proc. of OOPSLA*, 2002. To appear.

[9] jhotdraw software. `http://www.jhotdraw.org`.

[10] R. K. Keller, G. Knapen, B. Lagu, S. Robitaille, G. SaintDenis, and R. Schauer. The SPOOL design repository: Architecture, schema and mechanisms. *Adv. in Soft. Eng.: Topics in Evolution, Comprehension, and Evaluation*, pages 269–294, 2000.

[11] M. Kircher and P. Jain. Caching architectural design pattern. In *Tech. Rep. of Corporate Technology, Siemens AG*, Munich, Germany, 2002.

[12] D. Knuth. Literate programming. *Computer Journal*, 27(2):97–111, 1984.

[13] J. Korn, Y. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of java applets. In *Working Conf. on Rev. Eng.*, pages 314–325, 1999.

[14] R. Lougher and T. Rodden. Group support for the recording and sharing of maintenance rationale. *Soft. Eng. Journal*, pages 295–306, November 1993.

[15] LTCHUNK. `http://www.ltg.ed.ac.uk/index.html`.

[16] H. Müller and K. Klashinsky. A system for programming-in-the-large. In *Proc. of Int'l Conf. on SE*, pages 80–86, 1988.

[17] D. Parnas. Software aging. In *Proc. of Int'l Conf. on SE*, pages 279–287, 1994.

[18] L. Prechelt and C. Kramer. Functionality versus practicality: Employing existing tools for recovering structural design patterns. *J. of Universal Computer Science*, 4(12):866–882, Dec. 1998.

[19] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurren and Networked Objects*, volume 2. Wiley & Sons, 2000.

[20] M. Sefika, A. Sane, and R. Campbell. Monitoring compliance of a software system with its high level design models. In *Proc. of Int'l Conf. on SE*, pages 387–397, 1996.

[21] T. Systa, K. Koskimies, and H. Müller. Shimba – an environment for reverse engineering java softare systems. *Soft.: Pract. and Exp.*, 31(4):371–394, 2001.

[22] J. Vlissides and A. Alexandrescu. To code or not to code, part i and ii. C++ Report, March and June 2000.

[23] Zen CORBA ORB. `http://www.zen.uci.edu`.