A Study of Evolution in the Presence of Source-Derived Partial Design Representations

Vibha Sazawal, Miryung Kim, and David Notkin
University of Washington
Computer Science & Engineering
Seattle, Washington 98195-2350, USA
{vibha, miryung, notkin}@cs.washington.edu

Abstract

When performing evolution tasks, software engineers focus on both the low-level changes required and the effects those changes will have on the system's design. The Design Snippets Tool *generates partial design representations intended to help engineers address one design criterion:* ease of change*. In this paper, we describe a study in which participants used the Design Snippets Tool and other aids to perform a restructuring task focused on ease of change. Our findings describe how participants proceeded through the restructuring task and how they used the Design Snippets Tool. The results show that participants used the Design Snippets Tool for high-level tasks such as discovery of design problems, identification of restructuring goals, and confirmation of design improvements.*

1. Introduction

Software engineers intend to build and maintain software systems that meet not only functional requirements but also relevant *design criteria*. Common design criteria for software systems include ease of change, ease of understanding, testability, and robustness. When modifying existing code, software engineers focus on both the low-level changes needed and the implications those changes will have on the system's design. For example, software engineers examine whether changes will affect the system's adherence to relevant design criteria or whether the system should be restructured before changes are implemented.

Today's integrated development environments (IDEs) provide significant support for browsing code, navigating through code, and editing code. Software engineers have far less support, however, for the high-level design decision-making they perform as they investigate and modify existing code. Some software engineers gather low-level results from tools such as `grep` and then process it into design-level information. Software engineers also turn to low-tech tools such as whiteboards and notebooks.

Both low-level tools and low-tech tools have some advantages. Neither requires the use of a formal design notation; freedom from such notations is convenient when an engineer must divide attention between design-level ideas and code-level modifications. In addition, these tools remain useful when existing code is incomplete or in an inconsistent state. Nonetheless, these tools do not provide direct support for evaluating an existing system's design.

We have developed a new tool that provides better support for one design criterion: *ease of change*. If software is undergoing active evolution, then ease of change is essential [10]. Ease of change is a complex criterion that is affected by many properties of a software system. For the purposes of designing our new tool, we focused on three change-related design rules from the literature: (a) hide volatile implementation details behind an interface [14], (b) reduce coupling between modules [1], and (c) if volatile details must be revealed to some clients, then restrict clients who do not need privileged details to a narrower interface [3].[1] Our tool displays information relevant to compliance with these three design rules. More specifically, the tool analyzes code to generate partial design representations that we call *design snippets*.

In this paper, we describe an investigation into support for ease of change during software evolution. We present the results of a study in which participants were asked to perform a restructuring task. Participants could use standard IDE features, pen and paper, and our new tool that generates design snippets. In Section 2, we introduce the restructuring task that participants were asked to perform. This task is used as a running example when we describe the

[1] Other software engineering design rules also promote ease of change. For example, Parnas has defined rules for ease of extension and contraction [15]. Accommodation of these additional rules is a subject of future work.

Proceedings of the 7th International Workshop on Principles of Software Evolution (IWPSE'04)
1550-4077/04 $ 20.00 IEEE

Design Snippets Tool in Section 3. In Sections 4 through 6, we describe our study in detail and present results. Sections 7 and 8 discuss related work and conclusions respectively.

2. The restructuring task

We conducted the study presented in this paper *to learn more about design decisions made during evolution tasks* and *to explore what value our tool might provide when making decisions related to ease of change*. The study observes the use of our tool and other aids during a restructuring task. In this section, we introduce the restructuring task we asked study participants to perform. The next section introduces the Design Snippets Tool using this task as a running example.

The restructuring task motivates use of the design rules described in the Section 1. Most small tasks cannot realistically require application of all three design rules, so we limited the scope of our task to the first two design rules. The task was inspired by the description of the Strategy pattern in *Design Patterns* [5]. One application area for Strategy described in the book is text input validation.

We created a simple application that validated text input but did not use the Strategy pattern to implement input validation. Instead the application implemented validation in a more coupled and less flexible way. We call this application the "InputForm." The task given to participants is to restructure the InputForm application to make it easier to change.

2.1. The InputForm application

The InputForm application's GUI contains three text fields. A user inputs a date, a phone number, and a social security number. When a user clicks the "Enter" button, the application validates the entered inputs. If the format is invalid, the application displays the message "Format Error."

The InputForm application is implemented using six Java classes and one Java interface. `TestDriver` contains the Swing GUI code needed to provide an appealing look and feel to the application. It is not critical to the restructuring task. The `InputForm` class creates several instances of the `TextBox` class to populate the form. The remaining three classes – `PhoneFormat`, `DateFormat`, and `SSNFormat` – all implement the `Format` interface.

Key portions of the application related to the restructuring task are located in two methods of the `InputForm` class: `createTextBoxes` and `checkFields`. An excerpt of source code from these two methods is listed below.

```
public JPanel createTextBoxes() {
   JPanel panel = new JPanel();
   //create a date field
   dateField = new TextBox();
   dateField.setFormat(new DateFormat());
   JLabel dateLabel
    = new JLabel(date+":" + "YYYY/MM/DD");
   dateLabel.setLabelFor(dateField);

   //create a phone field
   TextBox phoneField = new TextBox();
   phoneField.setFormat(new PhoneFormat());
   JLabel phoneLabel
    = new JLabel(phone+": " + "(###)###-####");
   phoneLabel.setLabelFor(phoneField);

   // create a SSN field in the same way...

   // add each field to an array of
   // TextBox instances
   textboxes.add(dateField);
   textboxes.add(phoneField);
   textboxes.add(ssnField);

   // add textboxes to panel (omitted here)

   return panel;
}

public boolean checkFields() {

   Iterator i = textboxes.iterator();
   boolean result = true;
   while (i.hasNext()) {
      TextBox textbox = (TextBox)i.next();
      Format format = textbox.getFormat();
      String data = textbox.getText();

      if (format instanceof DateFormat) {
         DateFormat f = (DateFormat)format;
         if (textbox.check(f, data) == false) {
             result = false;
         }
      }
      if (format instanceof PhoneFormat) {
         PhoneFormat f = (PhoneFormat)format;
         if (textbox.check(f, data) == false) {
             result = false;
         }
      }
      if (format instanceof SSNFormat) {
         // ... analogous code here
      }
   }
   return result;
}
```

Despite the existence of three format classes, the code that performs the actual input validation is contained within the `TextBox` class. `TextBox` actually contains three methods named `check`, each one taking a different type of format as a parameter. `TextBox` also contains numerous static final constants that are used by the `check` methods. `InputForm` casts the `Format` instance associated with each `TextBox` instance and then calls one of the `TextBox.check` methods to execute the input checking.

This code contains numerous violations of the first two design rules mentioned in Section 1. First, implementa-

tion details about input validation are not adequately hidden from `InputForm`: `createTextBoxes` has knowledge of format strings (such as "YYYY/MM/DD") and `checkFields` makes assumptions about the types of formats that are returned from `TextBox.getFormat()`. Secondly, there is unnecessarily tight coupling among `InputForm`, `TextBox`, and all three format classes. Adding new kinds of text fields to this application is tedious and difficult.

2.2. The task and a plausible solution

The task was given to study participants as follows:

Restructure the InputForm application, with the anticipation that different kinds of text fields will be added in the future. For example, a planned future change involves adding text fields for "Zip code," "Credit card number," and "Expiration date." All three of the new fields will need input validation as well.

Your task is to restructure the existing application to make the software easier to change, given the knowledge of this upcoming change request.

One plausible solution to the task is to apply the Strategy pattern to the application. Input validation would be performed by the three format classes, not `TextBox`. `Format` would be changed into a class and would hold general constants likely to be used by multiple `Format` subclasses. `Format` subclasses would be responsible for preparing the format strings used in `createTextBoxes`. These changes would encapsulate input validation details behind the `Format` interface.

As stated in the Design Patterns book, a known feature (or flaw, depending on your perspective) of the Strategy pattern is that "clients must be aware of different Strategies" [5, page 318]. In other words, the `InputForm` must still know which of the `Format` subclasses to instantiate when creating a text field. Because `InputForm` is a simple application, this feature of Strategy may be acceptable. Alternatively, a restructuring may also include application of the Factory pattern. A well-implemented factory could encapsulate the creation of `TextBox` instances and their associated formats from the `InputForm` class and thus remove the coupling between `InputForm` and the `Format` subclasses.

3. The Design Snippets Tool

The Design Snippets Tool generates partial design representations (*design snippets*) from code. A design snippet is a partial, lightweight design representation that is displayed with an associated unit of code and is useful for design evaluation. The usage scenario assumes that software engineers will view code and design snippets at the same time or in the same small time frame. Co-display of design snippets and code creates a shared context that increases the comprehensibility of snippets and allows the software engineer to focus more easily on design details related to a unit of interest.

When evaluating a single unit of a codebase, engineers often need to consider the unit's relationships with other parts of the codebase. Design snippets assist in this task by providing both information about the current unit and relevant information about other parts of the codebase. In other words, design snippets are partial but not local. Design snippets offer a broader context that supports decision-making yet is sufficiently scoped to ease transitions between viewing code and snippets.

Snippets are lightweight in several senses. First, they are designed to be used concurrently with code. Second, the unit of code under consideration has a set of associated snippets that are automatically, quickly, and statically extracted from the code and kept up-to-date as the code is modified. (The analysis of the code is done statically.) Third, they are designed to be easy to integrate into existing software evolution processes.

The Design Snippets Tool is implemented as a plug-in to the Eclipse Java IDE [4]. Since Eclipse programmers edit Java files, the "unit" of code associated with each snippet is a Java file. The Design Snippets Tool displays the set of design snippets associated with the active Java file. Figure 1 shows a screenshot of the Eclipse IDE with the Design Snippets Tool plug-in. The snippets appear below the code.

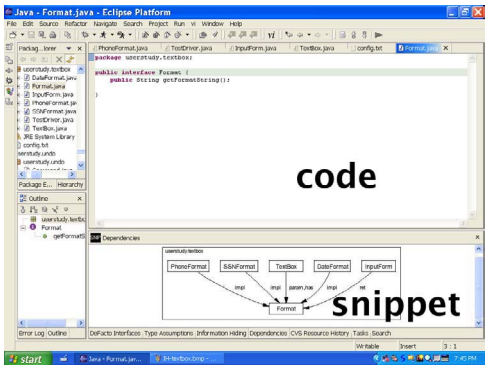

Figure 1. The Eclipse Java IDE with Design Snippets Tool plug-in

The Design Snippets Tool currently computes four design snippets, each supporting one of the three "ease of change" design rules described in Section 1. The *Information Hiding snippet* and *Type Assumptions snippet* support adherence to the first design rule (hide implementation details). The *Dependencies snippet* supports adherence to the second design rule (reduce coupling). The *De Facto Interfaces snippet* supports adherence to the third design rule

(restrict non-privileged clients to a narrow interface). As we describe each of the snippets, we will provide examples of the snippet views that are generated from files in the InputForm application.

3.1. Information Hiding snippet

The Information Hiding snippet provides information that helps the programmer evaluate the separation between interface and implementation. The snippet computes a view of the interface and implementation of each type defined in the associated Java file. The interface view lists superclasses, superinterfaces, and non-private member signatures. The implementation view lists private member signatures and "other classes used" by the class. The "other classes used" are neither parameters nor fields, but the implementation depends on them. These dependencies cannot be determined from perusal of the class declaration.

Using the Information Hiding snippet, a software engineer can identify relationships revealed by the interface and relationships hidden by the implementation. The snippet's display is organized for easy comparison of interface and implementation.

Figures 2 and 3 display screen shots of the Information Hiding views associated with TextBox.java. We present the interface view and the implementation view separately, but in the actual tool they appear side-by-side.

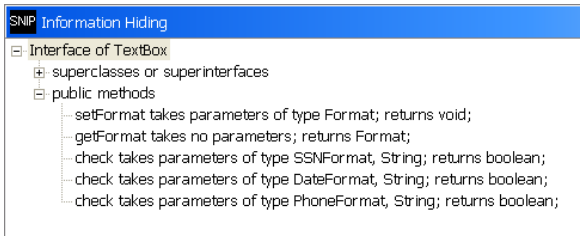

Figure 2. Information Hiding view for Textbox.java, interface portion

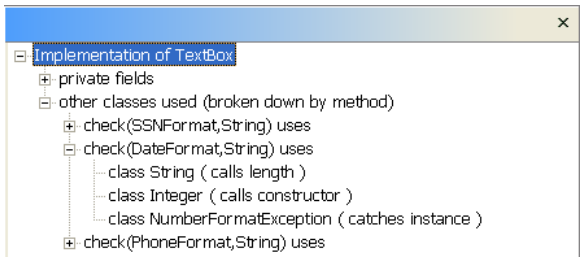

Figure 3. Information Hiding view for Textbox.java, implementation portion

3.2. Type Assumptions snippet

The Type Assumptions snippet lists casts of parameters and return values. These casts matter because the method signature is the interface between caller and callee. If a callee casts a parameter, then it makes an assumption about the data passed to it by the caller. If a caller casts a return value, then it makes an assumption about the data passed to it by the callee. These assumptions violate the separation between interface and implementation.

Figure 4 shows the Type Assumptions view associated with TextBox.java. The view shows that `InputForm` makes assumptions about the possible run-time types of the `Format` instance returned from `TextBox.getFormat`.

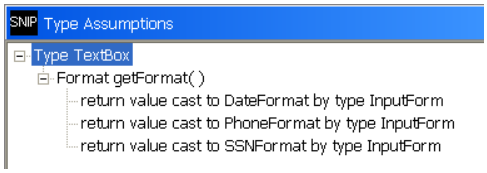

Figure 4. Type Assumptions view for Textbox.java

3.3. Dependencies snippet

The Dependencies snippet describes interclass relationships. Given a set of types T defined in the currently active file, the Dependencies snippet displays which types depend on T and which types T depends on. Edge labels indicate the cause of the source and sink's relationship.[2] Types are clustered into groupings based on Java packages, so dependencies that cross package boundaries are easily identified. Design Snippets Tool users can also choose to elide dependencies in order to focus on classes of interest. The Dependencies snippet is intended to help software engineers reduce coupling between modules.

Figure 5 shows an example of the Dependencies view associated with `InputForm.java`. Edge labels are associated with the edge to the left of the label. Dependencies to Java library classes have been elided.

3.4. De Facto Interfaces snippet

The De Facto Interfaces snippet helps the programmer identify the width of interfaces used by clients. A *de facto interface* [9] is the set of members actually used by a client. For each type defined in the currently active file, the De Facto Interfaces snippet reports a list of clients and the de

[2]For example, A "new" B means that A dynamically instantiates an instance of B. A full explanation of all edge labels appears elsewhere [17].

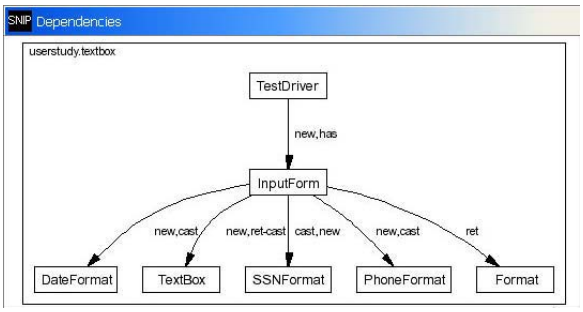

Figure 5. Dependencies view for InputForm.java

facto interface for each client. With the De Facto Interfaces snippet, software engineers can evaluate whether volatile design details are being properly restricted to privileged clients.

An alternative view of the De Facto Interfaces snippet reports a list of members and a set of clients for each member. This reordering is similar to a call graph. The call graph style may be more useful to software engineers when the third design rule (restrict non-privileged clients) is not immediately relevant.

Figure 6 shows the De Facto Interface views for PhoneFormat.java.

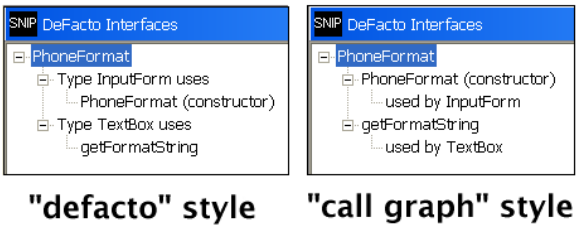

Figure 6. De Facto Interfaces views for PhoneFormat.java

4. Study

The purpose of the study was to observe use of the Design Snippets Tool and other tools during a restructuring task focused on ease of change. We wanted to see if the Design Snippets Tool provided value to study participants, and in particular whether the Design Snippets Tool helped study participants with design decision-making related to ease of change.

Any restructuring task is open-ended by nature, and we expected study participants to approach the task in different ways. In addition to understanding the value of the Design Snippets Tool, we were interested in two specific questions: *How did study participants use design snippets?* and *How did study participants decide what to change?*

4.1. Method

Eight subjects participated in the user study – two were software engineering practitioners from industry and six were graduate students in computer science. All eight participants were experienced programmers.

Sessions ranged in time from 60 to 90 minutes. All participants first read a set of tutorial slides. The tutorial introduced the first two design rules for ease of change mentioned earlier ("Isolate implementation details behind an interface" and "Reduce coupling between modules"). The tutorial then described the four design snippets. Only the call graph style of the De Facto Interfaces snippet was described. Study participants were permitted to ask questions as they read the tutorial and throughout the remainder of the session.

After reviewing the tutorial, users were given a short handout that described all the edge labels of Dependencies snippet graphs. Study participants could choose to review the edge label handout immediately or reserve it as a reference for later use.

Study participants were then given a functional description of the InputForm application. Upon review of the functional description, the study participants launched the Eclipse IDE and were shown the code for the InputForm application. The study participants were also shown the location of the four design snippet views in the Eclipse IDE.

Each user then completed three warm-up tasks designed to give them familiarity with both the Design Snippets Tool and the InputForm application. Example questions from the warm-up tasks include "What classes depend on `PhoneFormat`?" and "Who calls `TextBox.getFormat`?" The warm-up tasks were designed to be completed using the Design Snippets Tool, but users were free to find answers directly from perusal of the code or by using other tools available in the Eclipse IDE. Users were allowed to ask questions related to both the Design Snippet tool and other features of the Eclipse IDE.

After the warm-up tasks, study participants were given the restructuring task as described in Section 2.2. After the restructuring task, study participants answered a series of follow-up questions.

Each study session was attended by a facilitator and a notetaker. Study participants were encouraged to think aloud [11] during both the warm-up tasks and the restructuring task. For six of the eight participants, we recorded mouse and keyboard actions using screen capture software.

4.2. Two study session narratives

In this section, we present two study session narratives. In the next section, we will summarize our findings from analysis of all study session narratives. To help preserve anonymity, all participants will be referred to using female pronouns. The narratives were created by merging handwritten notes from the user sessions with screen capture data.

These two narratives illustrate two different usage scenarios with the Design Snippets Tool. Participant A is very familiar with the Eclipse IDE and used its functionality along with extensive review of the code. She used design snippets briefly during the start and end of the restructuring task. Participant B, in contrast, made extensive use of design snippets before modifying the code. Each narrative demonstrates ways in which design snippets can be incorporated into the evolution process.

Participant A. Participant A is a graduate student in computer science. She has used Java extensively over the last five years and has used the Eclipse IDE for one year. This participant finished the restructuring task very quickly compared to other participants, using less than 20 minutes.

Initial exploration and identification of design problems. When the participant began the restructuring task, she looked briefly at the Dependencies view of `InputForm`. The participant then went on to study the code for `InputForm`, `DateFormat`, and `TextBox`. While viewing TextBox.java, the participant saw declarations for the three check methods in Eclipse's Outline view.[3] The participant said aloud, "Oh ugliness!"

Solution approach. The participant then proceeded to move the check methods out of `TextBox` into the three format classes. Due to compiler errors generated by her changes, the participant moved `TextBox`'s fields into a new abstract base class called `AbstractFormat`.

After completing the restructuring task. At the end of the restructuring, the participant looked at the Dependencies view of `InputForm`. She noted that the view "shows no more casts." The participant stated that there was no more restructuring to do. The participant also said that it would take time to learn how to use design snippets effectively. She asked if she could install the design snippets tool on her personal machine so that she could learn to better incorporate it into her process.

Participant B. Participant B works for a medium-sized software company. She has used Java for over three years but has not used the Eclipse IDE previously. The participant spent 30 minutes on the restructuring task.

Initial exploration and identification of design problems. The participant began by reviewing the code of `InputForm`, the Information Hiding view of `PhoneFormat`, and the Dependencies views of `PhoneFormat` and `SSNFormat`. After this exploration, the participant stated that `InputForm` and `TextBox` were too coupled to the three format classes.

The participant went on to study more snippet views and code, including the Type Assumptions view of `TextBox` (which shows the casts of `getFormat`'s return value), the Dependencies view of `InputForm`, and the code for `InputForm.createTextBoxes`. At this point, the participant stated that `createTextBoxes` contains too much code and that `InputForm` should not need to change much when new text fields are added.

The participant then viewed the code and Dependencies views of `PhoneFormat` and `Format`, followed by another review of `InputForm` code. At this point, the user suggested that application of the factory pattern might remove the coupling between `InputForm` and the three format classes. The participant also proposed a new class that would encapsulate a `JLabel` instance, a `Format` instance, and a `TextBox` instance. `InputForm` would instantiate instances of this composite class instead of all three separately.

The participant moved on to the Dependencies view of `TextBox`. She stated that "`TextBox` depends on too much." She then used the De Facto view of `TextBox` to determine the caller of `Textbox.check`. The De Facto view led her to `InputForm.checkFields`.

Solution approach. The participant then ended her exploration and began restructuring the application. She converted `Format` to a class and moved validation-related constants from `TextBox` to `Format`. `TextBox`'s `check` methods were moved to the three specific format classes. The participant then returned to `InputForm.createTextBoxes` and replaced the magic strings (e.g. "YYYY/MM/DD") with calls to `Format.getFormatString`.

After completing the restructuring task. The participant stated that if she had more time, she would implement the composite class described earlier and then study the design snippet views to explore the effects of her changes. The participant also mentioned that the Type Assumptions and Dependencies views for `TextBox` helped her identify design problems.

5. Findings from study session narratives

In this section, we summarize findings from seven of the eight study sessions. To analyze study session narratives, we partitioned each session narrative into a series of episodes. Each episode consists of actions and statements made in the same time frame toward a single purpose. Use of the think-aloud protocol greatly assisted our ability to as-

[3] Eclipse's Outline view lists the members of all classes defined in the current Java file.

certain the purpose of participants' actions, and the process of building episodes enabled us to connect actions with intentions. We then reviewed the episodes for each participant to answer questions of interest.

We omitted one session from our analysis because the participant involved did not complete the restructuring task. This participant stated that the task as written suggested that only three new text fields would be added, and that three fields was not enough motivation for serious restructuring of the InputForm application. The participant proceeded to add a new text field to the application, but the task of adding a new text field is fundamentally different than the restructuring task that all other participants completed. This participant did not use design snippets.

Sections 5.1 and 5.2 summarize our answers to the two questions posed in Section 4.1. In Section 6, we discuss what our study suggests about the value of the Design Snippets Tool.

5.1. How did study participants use design snippets?

Seven study participants chose to use design snippets as they completed the restructuring task. We categorized these uses based on the actions and statements that immediately preceded and followed snippet use.

To identify design problems. Four participants used design snippets to identify design problems with the system. Design problems were identified from the Type Assumptions view of `TextBox`, the Dependencies view of `TextBox`, and the Dependencies view of `InputForm`. Study participants made comments such as "Oh, I see. It's annoying that `TextBox` explicitly depends on [format classes]."

To plan restructuring or identify restructuring goals. Three study participants used design snippets to plan their restructuring or to identify objectives. After seeing the Dependencies view for `InputForm`, one participant decided her restructuring should remove the "cast" edge label between `InputForm` and the format classes. The participant stated after the task that the undesirable edge labels gave her a goal. After reviewing design snippets and code, two participants drew Dependencies-like diagrams on paper that described the inter-class structure they wanted the restructured version to have.

To examine effects of changes on design. Three participants used snippets to examine the effects of their restructuring changes on the system's design. For example, two participants used the Dependencies view of `InputForm` to confirm that the "cast" edge label was no longer present. One participant used the Information Hiding view of `TextBox` to confirm that she had simplified `TextBox`'s public interface. That participant also used the Type Assumptions view of `TextBox` to confirm she had removed the casts of `getFormat`'s return value. One participant expressed interest in returning to design snippets to evaluate her changes but ran out of time.

To increase program understanding. Four participants used design snippets to assist in program understanding tasks. Often these program understanding tasks were related to determining the effects of changing part of the code. One participant used the Information Hiding view of `InputForm` to determine which `InputForm` method called the format classes' constructors. That participant also used the Dependencies view of `SSNFormat` to learn which classes depend on it. One participant used Dependencies views extensively to learn about the different classes in the program and their relationships to each other. Three participants used the De Facto Interfaces view to determine the callers of methods. One participant mentioned that class usage can be more important when restructuring than class definitions.

To discover relevant questions about the code. Finally, the design snippet views prompted two participants to ask questions about the code. For example, one participant asked, "Why are these fields in `TextBox`?" when looking at the Information Hiding view of `TextBox`. Another participant asked, "Where are `Format` instances casted?" when viewing the Dependencies view of `InputForm`.

5.2. How did study participants decide what to change?

Seven participants made changes in response to problems they found in the system. Participants differed in the way that they discovered problems, the way they characterized the problems, and the solution approaches they took in response to the problems. We present four approaches to problem discovery below.

Review of design snippets in the context of code. As discussed above, five of the participants used design snippets to identify problems or plan their solution approaches. These participants described problems in terms of classes being too coupled and return values being inappropriately casted. Sometimes the description of a problem and solution approach were combined; for example, one participant expressed the desire to restructure the code so certain edge labels would not appear in the Dependencies view of InputForm.java. All participants who studied design snippets also studied the code, and often additional problems or solution ideas came from review of the code. For example, one participant's study of casts in the Dependencies view of `InputForm` led to review of `InputForm.checkFields`. The participant then stated, "This is silly. Format classes should do their own checking."

Review of the code. Two participants identified problems solely through review of the code. Design snippets

may have helped them understand the code, but the code was their primary source for identification of problems and possible restructuring solutions. One participant identified a problem area by stating that it did not match her intuition. The solution approach was then to change the code so that it better matched how the participant thought it should work. Another participant looked at the three `check` methods in `TextBox` and diagnosed the problem as one of "trying to handle all the types [formats] in one class." She immediately proposed the creation of a `TextBox` class hierarchy, with one `TextBox` subclass for every format.

Use of other Eclipse features. Two participants identified problems using other Eclipse tools. For example, these participants realized that there were three `check` methods from Eclipse's Outline view. Both of these participants also studied the code and design snippets.

Compiler errors caused by previous changes. Once participants identified a solution path, some participants would immediately start making changes, while others would first consider the implications of these solution ideas on other parts of the code. For example, two of the participants, both of whom primarily used snippets to identify design problems, realized that their solution approaches would require the movement of validation-related constants out of `TextBox`. They made this realization before making any actual changes to the code. In contrast, four participants did not consider the validation-related constants in `TextBox` until they encountered compiler errors created when they moved the `check` methods out of `TextBox`.

6. Discussion

In this section, we discuss additional results that pertain to the value of the Design Snippets Tool. We recognize that only eight subjects participated in the study and that the restructuring task was a problem of our own choosing. As a result, we cannot generalize our results, nor can we claim to completely understand the benefits and costs of design snippets use. These limitations affect the external validity of our study. Nonetheless, we believe the study informs our understanding of how design snippets can be used to support design decision-making related to ease of change. The study can also inform our understanding of how the tool can be improved.

6.1. Co-viewing of code and design snippets

Earlier in the paper we suggested that design snippets and code are intended to be viewed in the same small time frame. The study provides evidence that supports this claim. All study participants who used design snippets used them in the context of reviewing or editing code. Some participants primarily worked with code and viewed design snippets infrequently, while others switched between code and design snippets more frequently.

Table 1 describes the context switches made by the seven study participants who used snippets. A context switch is a switch between one Java file and another Java file, a switch between a Java file and a design snippet view (or vice versa), or a switch between two design snippet views. For the purposes of this analysis, we did not consider other actions performed by users, such as using other Eclipse tools, running the application, or drawing diagrams. The first four columns list the number of switches performed by each participant (A-G). In the last column, we show the percentage of context switches that are code-to-snippet (code/DS) or snippet-to-code (DS/code). These numbers were computed from the screen recordings and our notes.

The table suggests that most study participants switched numerous times between design snippets and code. This evidence is promising, because our intended usage scenario assumes that software engineers will view design snippets as they view and modify code.

Table 1. Context switches by participants

	Context Switches				
	from design snippet to code	from code to design snippet	between two snippets	between two Java files	percent switches that are code/DS or DS/code
A	1	2	0	29	9%
B	3	3	9	14	21%
C	6	7	22	52	15%
D	2	2	2	18	17%
E	5	4	7	39	16%
F	9	9	5	25	38%
G	5	5	7	20	27%

6.2. Support for ease of change

The Design Snippets Tool was created to support decision-making related to ease of change. Did design snippets help participants with design-level decision-making, and specifically, did design snippets help with ease of change?

Seven of the eight study participants used design snippets to identify design problems, identify restructuring goals, or confirm that their changes had improved the system's design. These types of uses indicate that snippets may help with high-level design decision-making and evaluation of a system's design.

With regard to ease of change, four participants mentioned that design snippets led them directly to undesirable couplings or casts. Three participants mentioned that design snippets confirmed that their changes had removed casts or improved an interface. While few of these participants explicitly used the phrase "ease of change," most of them made comments that highlighted symptoms or solutions directly related to ease of change. We believe these comments indicate that design snippets helped participants evaluate and improve the system with regard to ease of change.

We should note that participants also discovered "ease of change" problems by reviewing the code. For example, three participants noticed duplication in `InputForm.createTextBoxes` after studying the code. One participant said, "If my goal is to make it easy to add text fields, I definitely want to make this [`createTextBoxes`] less repetitive."

When referring to duplication in `createTextBoxes`, one participant stated that "It's a bit deceptive if you start thinking that design snippets will show you all the problems with the code, because it does not." This comment is consistent with the role that code plays in our usage scenario. Design snippets cannot replace code review; instead, our view is that design snippets and code should be studied together. Using both design snippets and code, most study participants successfully identified and fixed problems related to ease of change.

6.3. Scope and availability of design snippet representations

Design snippets are partial representations. For example, the Dependencies view for `TextBox` displays only those classes that depend on `TextBox` or that `TextBox` depends on. Earlier in the paper, we suggested that the partial scope of snippets eases transitions between Java files and snippets. Table 1 suggests that study participants were indeed able to transition easily between Java files and snippets.

We also suggested earlier that the non-local information provided by snippets offers a broader context that assists decision-making. Anecdotal evidence from the study suggests that participants appreciated the non-local information. Participants' comments include:

- "Cool, I can get it over here." (The participant is referring to the casts visible in `TextBox`'s Type Assumptions view, even though the casts actually occur in `InputForm`.)

- "This [the Dependencies view] sort of reverses who uses a particular [class] . . . [it's the] reverse of [the] typical [view] . . . it is very useful." (This participant is describing how the Dependencies view displays the classes that depend on a class in addition to what the class depends on.)

- "Now this [De Facto Interfaces view] is useful [because it lists callers when callee code is active] . . . this seems to be a more natural interface for me." (The participant is comparing the De Facto Interfaces view to Eclipse's "Search References" right-click menu option [7].)

Study participants viewed snippets as they made changes to the code and when they had completed their changes to the code. Snippet views were updated as participants made changes, with no effort required on the part of the participant. One participant appreciated the real-time update of design snippet views, saying, "[The Dependencies view] is very helpful. I have used a UML visualization [tool] . . . it is inconvenient, it takes time to update every time the structure changes. This was convenient. I don't know of a better way to do this."

6.4. Possible tool improvements

Navigation from design snippets to code. Design snippets in their current form are passive. Most participants, however, attempted to navigate to different parts of the codebase by clicking on elements in snippet views. As one participant said, "I want to tie backward from the design snippet tools to the code." For example, navigation support for the Type Assumptions view could include clicking on entries to jump to locations of casts in the code.

Intra-module information. Three participants tried to use design snippets to determine the specific `TextBox` methods in which private `TextBox` fields were accessed. Since design snippets do not display intra-module dependencies, they did not answer this question. The three design rules that form the foundation for the Design Snippets Tool focus on inter-module relationships, not intra-module relationships. During the study, however, it became clear that solutions to inter-module problems often require that a module be torn apart. Intra-module information is very useful when moving pieces of a module to other places. One or more intra-module design snippets could also help software engineers meet design rules related to intra-module structure.

7. Related work

Model-driven development tools. Model-driven development tools, such as IBM's Rational Rose XDE [6] and Borland's Together [2], support the creation of both UML diagrams and code. These tools automatically generate code from UML diagrams and vice versa.

The main difference between these tools and the Design Snippets Tool is the form and role of design representations. Using a model-driven tool, software engineers can create large-scale UML models that describe the entire system. In contrast, the Design Snippets Tool provides partial design representations. These partial representations do not fully describe the system; instead, they support three ease-of-change rules. Design snippets exist to promote adherence to design criteria. Their intended users focus primarily on code but would benefit from design decision support as they modify code.

Program understanding tools. Tools such as Rigi [13] and SHriMP [18] visualize the structure of software systems. SNiFF+ [8] performs static analysis to identify references to symbols and visualize inter-module relationships. Lemma [12] supports navigation through control flow and data flow paths. Empirical studies of program understanding tools have been performed by Storey *et al.* [19] and von Mayrhauser and Lang [21].

The Design Snippets Tool differs from program understanding tools because of its explicit focus on ease of change. Design snippets can assist code comprehension, but the views presented are not intended to provide complete understanding. Instead, design snippets elide details unrelated to ease of change.

Design critics. Design critics [16] are design-support agents that automatically critique designs. One design critic tool, ArgoUML [20], has a set of built-in design rules for UML diagrams. Examples of rules include "package names should be written in lower case" and "circular compositions are not permitted."

The Design Snippets Tool also supports a set of rules (in this case, three rules related to ease of change). However, the Design Snippets Tool analyzes code, not design models. In addition, the Design Snippets Tool does not explicitly identify rule violations; instead, software engineers view design snippets in the context of code to manually assess the tradeoffs and decisions related to ease of change.

8. Conclusion

Design snippets are partial and lightweight design representations that help software engineers assess adherence to three design rules related to ease of change. In this paper, we present a study in which participants used design snippets and other aids to complete a restructuring task focused on ease of change. The results suggest that study participants used design snippets in the context of code for high-level tasks such as discovery of design problems and confirmation of design improvements. Participants also made decisions using knowledge gained from code review and low-level tool output (such as compiler errors). The study's findings increase our understanding of how design representations can be used in the context of evolution. We plan to use the study's results to guide future improvements to the Design Snippets Tool.

References

[1] G. Bergland. A guided tour of program design methodologies. *IEEE Computer*, October 1981.

[2] Borland Together. [http://www.borland.com/together].

[3] K. Britton, R. A. Parker, and D. Parnas. A procedure for designing abstract interfaces for device interface modules. In *Proceedings of the 5th International Conference on Software Engineering*, March 1981.

[4] Eclipse Foundation. Eclipse. [http://www.eclipse.org].

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[6] IBM Rational Rose XDE Developer. [http://www-306.ibm.com/software/rational].

[7] A. Kiezun. Java user development guide, 2004. [http://www.eclipse.org/documentation/main.html].

[8] M. Klaus. Simplifying code comprehension for legacy code reuse. *Embedded Developers Journal*, April 2002.

[9] W. Korman and W. Griswold. Elbereth: Tool support for refactoring Java programs, 1998. Technical report, University of California, San Diego Department of Computer Science and Engineering.

[10] M. Lehman and L. Belady. *Program Evolution: processes of software change*. Academic Press, 1985.

[11] C. Lewis. Using the 'thinking-aloud' method in cognitive interface design, 1982. Technical report RC9265, IBM T.J. Watson Research Center.

[12] R. G. Mays. Power programming with the lemma code viewer, 1996. Technical report, IBM TRP Networking Laboratory.

[13] H. Müller and K. Klashinsky. Rigi – a system for programming-in-the-large. In *Proceedings of the 5th International Conference on Software Engineering*, April 1988.

[14] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, December 1972.

[15] D. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, March 1979.

[16] J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Software architecture critics in argo. In *Proceedings of the 1998 Conference on Intelligent User Interfaces*, 1998.

[17] V. Sazawal. Dependencies snippet cheat sheet. [http://www.cs.washington.edu/homes/vibha/ds.html].

[18] M.-A. D. Storey, H. A. Müller, and K. Wong. Manipulating and documenting software structures. *Software Visualization*, 1996.

[19] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2–3), 2000.

[20] Tigris.org. Argouml. [http://www.argouml.org].

[21] A. von Mayrhauser and S. Lang. On the role of static analysis during software maintenance. In *Proceedings of the Seventh International Workshop on Program Comprehension*, 1999.