The Impact of Static-Dynamic Coupling on Remodularization

Rick Chern   Kris De Volder

University of British Columbia
2366 Main Mall
Vancouver, BC
Canada
{rchern, kdvolder}@cs.ubc.ca

Abstract

We explore the concept of *static-dynamic coupling*—the degree to which changes in a program's static modular structure imply changes to its dynamic structure. This paper investigates the impact of static-dynamic coupling in a programming language on the effort required to evolve the coarse modular structure of programs written in that language. We performed a series of remodularization case studies in both Java and SubjectJ. SubjectJ is designed to be similar to Java, but have strictly less static-dynamic coupling. Our results include quantitative measures—time taken and number of bugs introduced—as well as a more subjective qualitative analysis of the remodularization process. All results point in the same direction and suggest that static-dynamic coupling causes substantial accidental complexity for the remodularization of Java programs.

Categories and Subject Descriptors D.2.2 [*Design Tools and Techniques*]: Evolutionary prototyping, Modules and interfaces; D.3.3 [*Language Constructs and Features*]: Modules, packages; D.2.6 [*Programming Environments*]: Integrated environments; D.2.3 [*Coding Tools and Techniques*]: Object-oriented programming

General Terms languages, experimentation

Keywords remodularization, subject-oriented programming, hyperslices, static-dynamic coupling, refactoring, language design

1. Introduction

We consider two kinds of program structure: static and dynamic. Static program structure is concerned with how information in program text is organized in terms of document structure. Dynamic program structure is concerned with the structure of computations during program execution—for example, dynamic program structure in an object-oriented language corresponds to the structure of objects and their run-time interactions.

In the remainder of this paper, we refer to "remodularization" as the evolution of a program's coarse static modular structure. Language design can impact the complexity of remodularizing programs. In particular, some language designs can make it harder to move code around within the static modular structure without changing dynamic program structure; we call this *static-dynamic coupling*.

The key contribution of our paper is providing experimental results towards answering the following research question:

> *How and to what extent does static-dynamic coupling in a language impact the complexity of remodularizing programs?*

To investigate this question, we performed a series of remodularization case studies in two languages—Java [13] and SubjectJ. SubjectJ was designed to be similar to Java but have less static-dynamic coupling. We collected quantitative data—time taken and number of bugs introduced—and also performed a more subjective qualitative analysis of the remodularization process.

All results point in the same direction and suggest that static-dynamic coupling in Java causes substantial accidental complexity[1]. Quantitative results show that remodularizing in Java takes more time and results in more frequent introduction of bugs; qualitative results provide insight into *how* static-dynamic coupling in Java leads to accidental complexity.

Note that we do not claim SubjectJ itself as a novel contribution. SubjectJ is only a guinea pig. The ideal pair of guinea pigs would be two languages that significantly differ from each other in terms of static-dynamic coupling,

[1] The term "accidental complexity" coined by Brooks [2] means "complexity that arises in computer programs or their development process which is non-essential to the problem to be solved".

```
public class Counter extends JPanel {
    private int count = 0;
    private JButton button;
    private JLabel label;

    public Counter() {
        label = new JLabel(""+getCount());
        add(label);
        button = new JButton("Increment");
        add(button);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                increment();
            }
        });
    }
    public int getCount() {
        return count;
    }
    public void increment() {
        count++;
        updateDisplay();
    }
    private void updateDisplay() {
        label.setText(""+getCount());
    }
}
```

Figure 1. GUI and model code tangled.

but are identical in all other respects. We chose Java as the first guinea pig, because it has particularly strong static-dynamic coupling, and because it is a mainstream language for which it is easy to find existing code bases. To be close to an ideal second guinea pig, SubjectJ was *not* designed to be innovative, but rather to be very similar to Java, while removing what we perceived to be Java's greatest source of static-dynamic coupling.

The remainder of this paper is structured as follows. We start by presenting a concrete example illustrating that Java has strong static-dynamic coupling and how this complicates remodularization. In Section 3 we describe SubjectJ. Section 4 presents the case-study experiment comparing remodularization in Java and SubjectJ. Related work is reviewed in Section 6, and we provide concluding statements in Section 7.

2. Static-Dynamic Coupling Example

The following example illustrates the concept of static-dynamic coupling, and shows how it complicates Java remodularization. More specifically, we will argue that static-dynamic coupling compels Java developers to adopt a solution that modifies dynamic program structure, even if they only wanted to change static structure. The complexity of the dynamic structure transformations makes them hard to support by refactoring tools, and the developer must resort to an error-prone "cut-and-paste and fix compiler errors" approach.

Figure 1 shows a Java "Counter" program, which has a single "Increment" button and displays the number of times this button has been pressed.

For presentation purposes, this example is deliberately simplistic, but its structure is representative of many realistic Java GUIs. The code that maintains the counter state is tangled with the GUI code. This is not good modularity, but nevertheless many GUIs are initially written in this way [25]. The simplicity of the example makes for clearer presentation, but may make the "problems" we are about to discuss seem trivial. We ask the reader to bear in mind that, in more realistic programs, the scattering and tangling of UI code could span across multiple classes of considerable complexity.

Now, suppose we wanted to remodularize this code to separate the model code from the GUI code, creating a clear interface between the two code modules, while preserving external program behavior. A typical way to achieve these remodularization goals is to implement a model-view-controller architecture, with a listener registration protocol on the model class(es). The result is shown in Figures 2 and 3.

```
public interface CounterListener {
    public void valueChanged();
}
public class CounterModel extends JPanel {
    List<CounterListener> listeners = new ArrayList<CounterListener>();
    private int count = 0;

    public CounterModel() {}

    public void increment() {
        count++;
        notifyListeners();
    }
    private void notifyListeners() {
        for (CounterListener l : listeners)
            l.valueChanged();
    }
    public void addListener(CounterListener l) {
        listeners.add(l);
    }
    public int getCount() {
        return count;
    }
}
```

Figure 2. A separate model with a listener infrastructure

```
public class Counter extends JPanel implements CounterListener {
    private CounterModel count;
    private JButton button;
    private JLabel label;

    public Counter() {
        count = new CounterModel();
        label = new JLabel(""+count.getValue());
        add(label);
        button = new JButton("Increment");
        add(button);
        button.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                count.increment();
            }
        });
        count.addListener(this);
    }
    public void valueChanged() {
        label.setText(""+count.getCount());
    }
}
```

Figure 3. GUI code as a client of the model

The type of transformation needed here—implementing a design pattern—cannot easily be broken down into refactoring steps supported by a typical IDE. Therefore, typically an ad-hoc "cut-and-paste and fix compiler errors" process will be used. This process is tedious, unpredictable, and error prone.

Recall however that our original intent was to change the coarse static program structure. As such, the implementation of a listener infrastructure, which is a transformation of dynamic structure, was strictly speaking uncalled for. At the same time, the listener implementation seems to be the main source of complexity.

To be clear, we do not imply that implementing a listener might not be useful, only that it was not an explicitly stated goal *in this example*. Even if we might eventually want to move to a listener based solution, being forced to adopt a particular implementation earlier than needed is poor incrementality. As such, we would like to avoid its complexity, or at least postpone it until it is really needed.

Unfortunately, in Java the complexity is unavoidable because of strong static-dynamic coupling: Java imposes structural limitations which imply that the remodularization goals cannot be met without changing dynamic structure. The main reason is that in Java, each class declaration is contained entirely within one `.java` file. This constraint makes it as good as impossible to separate model code from GUI code without creating separate runtime objects.

3. SubjectJ

In this section we provide an overview of SubjectJ, focussing on the aspects that are relevant for this paper. For a more detailed description of the SubjectJ language and tools we refer to [6].

Note that we do not claim SubjectJ itself as a novel contribution; it only serves as a guinea pig. To be close to an ideal guinea pig, it was designed explicitly to be very similar to Java, while relaxing what we perceived to be Java's greatest source of static-dynamic coupling: the constraint that a class's declaration must be wholly contained within a single Java source file. Thus, SubjectJ's primary design goal is to allow classes to be split across coarse-grained static modules. To be meaningful, we believe this goal should be achieved without diluting the module concept. This is captured by the second design goal—that SubjectJ modules should have well-defined explicit interfaces that allow hiding implementation details from other modules.

Finally, we adopted two pragmatic design goals. The first pragmatic goal was that Java programs should be valid SubjectJ programs. This allows experimenting on the same realistic Java code bases for both Java and SubjectJ remodularization case-studies. The second goal was that SubjectJ syntax should not radically break existing Java tools in the Eclipse[2] IDE. This allows the use of Eclipse tools with both Java and SubjectJ, so that SubjectJ and Java are comparable in terms of realistic IDE support.

[2] `http://eclipse.org/` verified December 2007.

```
@Shared public class Counter extends JPanel {
    private JButton button;
    private JLabel label;

    public Counter() {
        label = new JLabel(""+getCount());
        add(label);
        button = new JButton("Increment");
        add(button);
        button.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                increment();
            }
        });
    }
    @Import public int getCount();
    @Import public void increment();

    @Export private void updateDisplay() {
        label.setText(""+getCount());
    }
}
```

Figure 4. Version of Counter in the "UI" subject's source tree.

The design of SubjectJ builds heavily on the ideas of subject-oriented programming [17, 28] and multi-dimensional separation of concerns [32]. SubjectJ is essentially a "light" variant of Hyper/J [31], supporting only two dimensions of concern (class and subject) and a single built-in composition rule. However, SubjectJ modules have explicitly defined interfaces and can encapsulate implementation details. In contrast, Hyper/J hyperslices as described in [31] do not distinguish between exported and non-exported declarations, and thus do not truly encapsulate implementation details by hiding declarations from other hyperslices[3].

In the following sections, we describe in more detail the modular units of SubjectJ, and overview the tool support for transforming source code in SubjectJ. We then illustrate how SubjectJ can be used for remodularizing existing Java programs.

3.1 Modular Units in SubjectJ: Subjects

SubjectJ allows decomposing a Java program into modular units called "subjects". A subject is a partial Java program, with a well-defined public interface. Figures 4 and 5 show an example of the SubjectJ code we might end up with after separating the code from Figure 1 into two subjects, one for the UI related code (Figure 4) and one for the rest (Figure 5). Note that due to the simplicity of this example, each subject only contains declarations for a single class. In general however, a subject can contain multiple partial or complete classes from different Java packages, and is thus represented by a source tree with multiple `.java` files.

[3] Although the composition tools presented in [31] do not support true encapsulation of hyperslices, the limitation is not a conceptual constraint of the Hyper/J language, and is addressed by the composition engine described in [18].

Annotation	Attached to	Meaning
`@Export`	Field, Method, Constructor	The signature of this declaration can be imported by other subjects.
`@Import`	Field, Method, Constructor	The signature of this declaration must be exported by another subject. This annotation should only be attached to method declarations without a body, and field declarations without an initializer.
`@Shared`	Class	The class header (including "extends" clause, but not necessarily all "implements" clauses) can be shared with other subjects. "Implements" clauses are shared automatically if the corresponding interface is shared.
`@Shared`	Interface	The interface header (but not necessarily all "extends" clauses) can be shared with other subjects. "Extends" clauses are shared automatically if the corresponding interface is shared.

Table 1. Annotations defining a subject's interface

```
@Shared public class Counter extends JPanel {
    private int count = 0;

    @Export public void increment() {
        count++;
        updateDisplay();
    }
    @Export public int getCount() {
        return value;
    }
    @Import private void updateDisplay();
}
```

Figure 5. Version of Counter in the "OTHER" subject's source tree.

SubjectJ syntax is essentially Java syntax with custom Java 1.5 annotations. The purpose of the annotations is to define a subject's public interface; an overview of these annotations is shown in Table 1. The public interface of a subject is independent from Java's public/private modifiers, and is scoped relative to subjects rather than classes. When subjects are composed, we verify whether their interfaces are consistent with each other.

The annotations make it explicit how a subject depends on other subjects, and also what parts of the subject are visible to other subjects. For example, in Figure 4 and Figure 5 the `@Shared` annotation attached to the `Counter` class signifies that the declaration of the `Counter` class may be shared by other subjects. This places the name "Counter" conceptually in a public global namespace, allowing multiple subjects to refer to it and contribute their own field, constructor and method declarations to the class. Similarly, an `@Export` placed on a member means the signature of the member declaration is visible to other subjects, while an `@Import` on a member means the subject depends on the signature of the member declaration being made available by an `@Export` from another subject.

Note that although the `Counter` class is shared, this does not mean that all of its members must be shared. For example, the `button` and `label` fields are not exported and so they are private to the UI subject. This means that other subjects are not allowed to have (direct) dependencies on these fields, even though they may contribute code to the same class.

Only subjects that are *declaratively complete* [32] are considered valid. Declarative completeness means that a subject needs to include at least the signatures or headers of anything it depends on. This constraint ensures that a subject, although not a complete Java program, is relatively self-contained and can be understood and worked on in relative isolation. Declarative completeness also strengthens the notion of subject interfaces because it forces any dependency a subject has on other subjects to be explicitly represented by `@Import` or `@Shared` annotations.

The reader may have noticed our asymmetric treatment of "extends" clauses in classes versus interfaces, and the unnecessary `extends JPanel` clause in Figure 5 that is a consequence of this. Conceptually, SubjectJ breaks up Java classes into smaller syntactic units and allows these units to be rearranged into subjects. To decide on the granularity of these units, we performed some preliminary experiments remodularizing Java code into SubjectJ subjects. We only broke up a unit if we encountered scenarios where separating the unit across subjects would help achieve a remodularization goal from our preliminary experiments. So, we decided to allow individual separation of "implements" clauses (and "extends" clauses for interfaces), because we observed that entire interfaces were often irrelevant to a subject, but we decided not to allow "extends" clauses to be separated from their class headers, because we did not encounter scenarios in our preliminary experiments where a class declaration needed to be separated from its superclass. We *did* encounter such scenarios later on while performing the case study experiment described in Section 4; but, having already performed a number of case studies at this time, we decided not to remove the constraint on separating "extends" clauses. Although this constraint clearly impedes the movement of code undesireably, SubjectJ still legitimately serves as a language with looser static-dynamic coupling than Java in our case study experiments.

3.2 SubjectJ Tools

SubjectJ provides a number of tools for the purpose of editing, compiling, running and refactoring SubjectJ programs.

Figure 6. Overview of SubjectJ tools—Decompose, Compose, and Checker.

```
@Subject({"UI","OTHER"})
public class Counter extends JPanel {

    private int count = 0;

    @Subject("UI") private JButton button;
    @Subject("UI") private JLabel label;

    @Subject("UI") public Counter() {
        label = new JLabel(""+getCount());
        add(label);
        button = new JButton("Increment");
        add(button);
        button.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                increment();
            }
        });
    }
    @Export("UI") private int getCount() {
        return count;
    }
    @Export("UI") public void increment() {
        count++;
        updateDisplay();
    }
    @Subject("UI") @Export("OTHER")
    private void updateDisplay() {
        label.setText(""+getCount());
    }
}
```

Figure 7. Composed Java program with "Tracking" annotations.

Figure 6 shows an overview of these tools. The following subsections describe each of the SubjectJ-specific tools in more detail.

3.2.1 Compose

The Compose tool is conceptually very simple: it computes the union of all its input subjects, linking any `@Import` to a corresponding `@Export`, and generates a Java program. This Java program can then be compiled and run with standard Java tools.

The Compose tool also checks that subject interfaces are being respected. For example, consider two independent developers working on the UI and OTHER subjects. Suppose both added a helper method with an identical signature to the `Counter` class. Lacking appropriate `@Import` and `@Export` annotations, these methods should be considered hidden within their respective subjects. The fact that they would be mapped onto the same method declaration in the composed Java program violates this intent and is therefore a composition conflict. The current implementation of the Compose tool checks for conflicts and issues error messages. A more sophisticated implementation could automatically resolve such conflicts by renaming hidden declarations.

3.2.2 Decompose

The Decompose tool is the inverse of the Compose tool. The Compose tool inserts subject "tracking" annotations into a composed program. These annotations track what subjects the code in the composed program belonged to. The Decompose tool uses this information to invert the compose operation.

An overview of the tracking annotations is shown in Table 2. Figure 7 shows the result of composing the UI subject from Figure 4 and OTHER subject from Figure 5. Tracking annotations mark the code in the composed program with the names of the subjects they belong to. Note that different types of annotations apply to different parts of the code they are attached to. For example, the `@Export` annotation on `updateDisplay` indicates that its signature is exported to the OTHER subject, while the `@Subject` annotation on `updateDisplay` indicates that its body belongs only to UI.

Also note that an annotation like `@Export("UI")` implies both that the declaration it is attached to is imported by the UI subject and exported by whatever subject it belongs to. This is because composition fuses imports and exports into a shared declaration. A further detail to note is that any declaration that doesn't have an explicit `@Subject` is treated as having a `@Subject("OTHER")` annotation. Thus, the annotation on `getCount` implies that it belongs to OTHER (and is exported to UI). Finally, the `@Implement` annotation, which is not used in this simple example, marks individual "implements" clauses.

3.2.3 Checker

The Checker tool accepts a Java program, marked up with annotations as if produced by the Compose tool, and verifies whether the mark-up represents a valid decomposition of the Java program into subjects. This tool is intended to support use cases where the Decompose tool is run on "composed" programs that were produced or modified by the developer rather than the Compose tool.

Essentially, the Checker tool checks for declarative completeness. This is a relatively straightforward static analysis, verifying whether code marked as belonging to any given subject has static dependencies on declarations not included or imported in the subject.

Optionally, the Checker tool can automatically insert annotations to include the missing code. This is useful support for use cases involving user-generated or edited "composed" programs. Essentially, it allows a developer to only annotate the fields, methods, and constructors that belong to a given subject, and let the Checker tool infer proper subject interfaces from static dependencies.

3.3 Implementation

The SubjectJ Decompose, Compose and Checker tools are implemented as a single Eclipse plugin. Its implementation consists of approximately 4000 lines of code, of which ap-

Attached to	Meaning
`@Subject(`$S_1, \ldots$`)`	
Class	The class header (including "extends" clause but not necessarily "implements" clauses) belongs to subjects S_1,...
Interface	The interface header (not necessarily including "extends clauses") belongs to subjects S_1,...
`@Subject(`S`)`	
Field	The field (signature plus optional initialer) belongs to subject S
Method, Constructor	The signature and optional body belong to subject S
`@Export(`S_1`, ...)`	
Field, Method, Constructor	The declaration's signature is imported by subjects S_1,... and is exported by the subjects the declaration belongs to.
`@Implement( @Mapping`(key=S_1, values = I_1^1,I_1^2,...),...)	
Class	For each subject S_i, only the "implements" clauses for interfaces I_i^1, I_i^2, ... belong to S_i. For subjects not explicitly mapped, implicitly include all "implements" clauses.
Interface	Like for a class, but applies to "extends" instead of "implements" clauses.

Table 2. "Tracking" annotations that allow decomposing Java programs into subjects.

proximately 300 lines implement a rudimentary SWT UI that lets the user launch the SubjectJ tools from within Eclipse.

The running times of the tools are approximately linear to the size of the code base to which the tools are applied. Using a PC with a 2.13 GHz Intel Core 2 processor and 1GB of memory, we ran the tools on the largest code base in our case studies, with 70833 lines of code (see Section 4). The Decompose and Compose tools each required approximately 1 minute to run, while the Checker tool required approximately 2 minutes to run.

3.4 Remodularizing Java Programs with SubjectJ

In this section, we describe the typical way in which the SubjectJ tools just discussed would be used in the use case corresponding to our remodularization case studies: remodularizing an existing Java code base into two subjects. Variants of this use case to further remodularize a program already divided into subjects can also be supported, but discussion of them is not relevant to our experiment.

Reconsider the Java program from Figure 1. Suppose we wanted to remodularize this, separating all UI related code into its own UI subject and keeping the remaining code in the OTHER subject. To do this, we basically need to insert appropriate annotations to mark UI related code in the Java program, and then run the Decompose tool. The Checker tool can be used to assist in this process.

For our example it might go as follows. We start by placing four `@Subject("UI")` annotations: on the constructor; the `label` and the `button` fields; and the `updateDisplay` method. Recall that everything not explicitly annotated is treated as belonging to the OTHER subject. However, to make this a valid decomposition, we need more annotations to describe the interface between the UI and OTHER subject. We run the Checker tool to infer the needed annotations, producing the mark-up shown in Figure 7. Running the Decompose tool on this produces the remodularized program shown in Figure 4 and Figure 5.

4. Experiment

We now present our case study experiment, aimed at addressing our research question, "How and to what extent does static-dynamic coupling in a language impact the complexity of remodularizing programs written in that language?". In our experiment, we compared remodularization complexity between Java and SubjectJ, a language designed for reduced static-dynamic coupling compared to Java.

To compare remodularization complexity between Java and SubjectJ, we carried out a series of 8 case studies on 8 different open source Java software packages. Each case study involved performing a remodularization task twice—once in Java and once in SubjectJ. We then compared the results to each other. To quantify complexity, we measured the time taken to perform the remodularizations, and counted the number of bugs introduced while performing each remodularization. We also performed a qualitative analysis focused on finding anecdotal evidence of accidental complexity.

Qualitative and quantitative results complement each other. The quantitative results provide reasonably objective evidence that remodularization is easier in SubjectJ, while the qualitative results provide insight into *how*, by providing concrete examples of the problems encountered in the Java remodularizations.

We start by presenting our experimental setup in more detail in Section 4.1. We discuss the threats to validity of our experiment in Section 4.2. Then we present the quantitative results in Section 4.3, followed by the results of the anecdotal analysis in Section 4.4. Conclusions and limitations of the experiment are discussed in Section 4.5.

4.1 Experimental Setup

For each of the 8 case studies in the experiment, we selected a different open source Java code base. Table 3 has an overview of the selected code bases. We also defined for each case study a remodularization, which required the identification and separation of a certain subset of the existing code base, characterized by the high-level description shown under "Code to Separate" in Table 3. We defined an acceptable remodularization to be the creation of either a Java package or a SubjectJ subject containing all and only the identified source code (with any required interface code).

The 8 case studies were performed by the first author of this paper (who we call "the programmer"), in the order in which they are shown in Table 3. With the exception of the JHotDraw code base, he was unfamiliar with the selected code bases prior to performing the experiment.

For each case study the remodularization was performed twice consecutively, once using Java and once using SubjectJ. For the first 4 case studies the SubjectJ remodularization was performed first; for the last 4, the Java remodularization was performed first. The programmer was not allowed access to any data or results produced during the first remodularization process, while performing the second. The programmer was otherwise allowed to use all available tools, including automated Java refactoring tools in an installation of Eclipse 3.2.2. Since SubjectJ was designed not to break existing Java tools, Eclipse already provides some support for working with SubjectJ. However, Eclipse's Java browsing and refactoring tools do not understand the semantics of SubjectJ annotations. To bridge this gap and provide browsing and refactoring support for SubjectJ programs comparable to the level of support Java programs receive from the Eclipse JDT, we did two things. First, we installed version 3.1.13 of the JQuery [21] plugin and customized it to provide some rudimentary support for browsing Java elements marked with `@Subject` and `@Export` annotations. Second, we allowed the SubjectJ programmer to use the SubjectJ tools as described in Section 3.2. Used in this way, the Checker and Decomposer together function as simple refactoring tools for moving declarations from one subject to another without breaking static dependencies. This provides functionality similar to the Eclipse "Move Method" refactoring, but with SubjectJ subjects.

For each remodularization performed, we measured the total time needed to complete the task, made detailed notes of the steps performed during the remodularization process, and saved a copy of the code base upon completion of the task. The time data is presented and analyzed in Section 4.3. The notes and saved code are the basis for the more qualitative results presented in Section 4.4.

4.2 Threats to Validity

Our central research question is, "How and to what extent does static-dynamic coupling in a language impact the complexity of remodularizing programs written in that language?". In this section, we discuss the construct, internal, and external validity of our experiment with respect to this research question.

4.2.1 Construct Validity

We are interested in the theoretical constructs of our central research question—static-dynamic coupling and remodularization complexity. Construct validity describes how well the variables and observations in our experiment map to these theoretical constructs. There are two sources of threats to the construct validity of our experiment. One is that the variable we designed to vary in our experiment (programming language) is not precisely the theoretical cause we wish to understand (static-dynamic coupling). The other is that the set of outcomes we observe (remodularization time, number of bugs, and anecdotal evidence) is not precisely the theoretical effect we wish to understand (remodularization complexity).

First, our central research question concerns the impact of *static-dynamic coupling*. To understand the impact of static-dynamic coupling in our experiment, we compare two programming languages—Java; and SubjectJ, a language designed to have looser static-dynamic coupling than Java. A possible threat to construct validity is that SubjectJ does not *in fact* have looser static-dynamic coupling than Java. Another possible threat to construct validity is that SubjectJ and Java differ in more than just static-dynamic coupling, and that these differences can impact remodularization complexity. We have attempted to address these threats by explicitly identifying and removing Java's greatest source of static-dynamic coupling from SubjectJ, while otherwise designing the language to be very similar to Java. As described in Section 4.1, we also tried to design SubjectJ tool support to be comparable to Java tool support in Eclipse. However, the SubjectJ and Java tools are inevitably different. Since static-dynamic coupling in a language is partly reflected by its tool support, it is difficult to assess the extent to which tool differences are a threat to construct validity.

Secondly, our research question concerns remodularization *complexity*. To assess the extent of remodularization complexity in our experiment, we measured the time taken to complete each remodularization task, and the number of bugs produced during remodularization tasks. A possible threat to construct validity is that these measurements are not general indicators of complexity. However, to complement our reasonably objective quantitative measurements, we provide the qualitative observations of Section 4.4—which concern aspects of complexity that are more difficult to objectively measure. Our qualitative and quantiative results are

Application and Description[†]	LOC	Code to Separate
Tetris—game of Tetris `http://cslibrary.stanford.edu/112/`	1036	GUI handling.
TyRuBa—logic programming language `http://tyruba.sourceforge.net/`	22116	Storing and persisting "facts" used by the language.
JHotDraw—simple drawing application `http://www.jhotdraw.org/`	14611	All functionality for creating and modifying text figures.
Chinese Chess—game of Chinese chess `https://chinese-chess-xiang-qi.dev.java.net/`	3073	Logic for AI opponent.
MineRay—game of minesweeper `https://mineray.dev.java.net/`	3478	Logic for populating map with mines.
DrawSWF—simple animation application `http://drawswf.sourceforge.net/`	7540	All functionality for creating and modifying text figures.
FindBugs—Java source code bug finder `http://findbugs.sourceforge.net/`	70833	Saving of bug analysis results.
JChessBoard—game of chess `http://jchessboard.sourceforge.net/`	6190	GUI handling.

[†] All website references verified February 2008.

Table 3. Overview of selected code bases and corresponding refactoring tasks for case studies. LOC is the total number of non-blank and non-comment lines in the code base.

consistent with each other, together providing a more confident indicator of remodularization complexity.

4.2.2 Internal Validity

There are threats to the internal validity of our experiment–i.e., alternative causes (other than difference in programming language) that could affect our observed outcomes. One source of threats comes from learning effects. By performing the same remodularization task twice, there is bias that comes from knowledge gained by the programmer during the first iteration of the task. Our experimental setup addresses this by having the programmer perform half of the remodularizations using Java first, and half using SubjectJ first. However, the strength of the learning bias possibly varies depending on which language is first used to perform a remodularization, and there is likely a learning bias based on the order the 8 remodularizations were performed in the experiment. Our analysis in Section 4.3.1 considers how these threats limit the scope of our conclusions.

Another threat comes from the comparability of the 8 remodularization tasks given to the programmer. We address one threat in this respect, by limiting all remodularization tasks to be of the same type—identification and separation of a given functionality from a given code base. We also assign similar functionalities between the set of remodularizations performed using Java first, and the set of remodularizations performed using SubjectJ first. In particular, both sets of remodularizations require separation of GUI handling, data persistence, text figure creation and modification, and adversarial AI logic (population of game maps, or chess player opponent) functionalities.

A potential threat is the difference in code bases associated with each remodularization task. In particular, the code bases for Java-first remodularizations have more lines of code in total than the code bases for SubjectJ-first remodularizations. It is possible that this difference could introduce bias, but an analysis in Section 4.3.1 of our quantitative results suggests that this bias is limited for our experiment.

Finally, a social threat to internal validity comes from the use of the first author of this paper as the programmer who performs the remodularizations in the experiment. While the programmer strived to not allow his role as an author bias his performance, we concede that such bias is nevertheless possible, and recognize this threat as a limitation of our experimental setup.

4.2.3 External Validity

External validity involves the generalizability of our experiment results. Our central research question concerns the general impact of static-dynamic coupling in languages on remodularization complexity. In our experiment however, we compare only two specific languages (SubjectJ and Java) that differ in static-dynamic coupling, and remodularization complexity is measured based on tasks of a specific type (identification and separation of a certain subset of an existing code base) given to one programmer (the first author of this paper). Our results provide only one set of data, and may not be generalizable to other languages, tasks, and programmers.

In particular, our results may not be generalizable to languages that have static-dynamic decoupling at the sub-method granularity, such as AspectJ [23]. Even though our results suggest that inter-type declarations probably substantially reduce remodularization complexity, there is little grounds to draw similar conclusions about pointcut advice because these features operate at a finer level of granularity, and have no counterpart in SubjectJ.

In addition, the remodularization tasks in our experiment were intentionally selected to be of a specific type, and as such, determining the impact of static-dynamic coupling on other types of remodularization tasks (e.g. tasks based on concerns other than high-level software functionality) would require further investigation.

Finally, a significant threat to external validity is that all the remodularization tasks were performed by a single programmer—the developer of SubjectJ. Although the programmer was experienced with both the SubjectJ and Java languages and their refactoring tools, it is possible that he was better at using SubjectJ tools than Java tools. Also, different programmers could apply different programming styles and problem solving techniques to the same remodularization tasks. Experiments on larger samples of programmers would produce more generalizable results.

4.3 Quantitative Results

For each remodularization task, we measured the total time taken to complete the task, and recorded the bugs introduced while performing the task. Taking into account potential learning biases and other threats to internal validity described in Section 4.2.2, our results suggest that remodularization takes substantially less time in SubjectJ than in Java. The programmer also introduced significantly more bugs while remodularizing in Java than in SubjectJ.

4.3.1 Time Results

Table 4 provides an overview of time data, listed in the order the case studies were performed. The "SubjectJ" and "Java" columns show the time taken to perform the refactoring task in the respective language. The "Difference" column shows the difference between SubjectJ and Java times given as a percentage of the SubjectJ time. A positive difference indicates that the Java remodularization took more time. A negative difference indicates SubjectJ took more time.

The table is divided in two sections, based on which remodularization was performed first, implying a different learning bias caused by performing the same task twice. Note however, that the SubjectJ time is always placed in the first column. Similarly, the difference column is always computed relative to the SubjectJ time. This is *not* to suggest both sections should be interpreted in the same way, but to facilitate contrasting the numbers in both sections to each other.

In the top half of the table, the time differences are expected to be biased negatively, due to performing the SubjectJ remodularization before the Java remodularization for each case study. Despite this negative bias, the top half of the table shows a positive trend in time differences: two cases yield mildly positive differences, one case a strong positive difference, and one case yields a mildly negative difference.

The time differences in the bottom half of the table are expected to be biased positively, so a positive trend in this half is not surprising. However, the differences are consistently and significantly greater than in the top half the table. This suggests that the learning advantage for the second iteration of a task is substantial, and the negative bias on the top half of the table is strong.

We can compute an aggregate score that is not particularly weighted towards either half of the table. The two halves of the table have different biases, but time spent remodularizing was almost equal in the two halves (63.4 hours vs. 66.9 hours). Combining the 8 case studies, a total of 44.6 hours were spent using SubjectJ, and 85.7 hours were spent using Java. Thus, over the entire course of our experiment, about 92% more time was spent remodularizing in Java than in SubjectJ.

As we mentioned in Section 4.2.2, internal validity threats limit the scope of our conclusions. First, in light of our qualitative results (see Section 4.4), we believe that there is a stronger learning bias when the Java remodularization is performed first, as we observed that remodularizing in Java requires a more in-depth understanding of the code base. We also believe that the programmer may have become more fluent in performing case studies during the course of the experiment, as he became more familiar with remodularization techniques (e.g. "separation strategies" for Java).

It is also possible that bias could be introduced by the code bases in the bottom half of the table having more total lines of code than the code bases in the top half. However, this bias is likely limited in our experiment, as we do not see a significant correlation between lines of code and percentage differences in remodularization time. For instance, the 4 differences in the bottom half of the table range from 183% to 325%, but the case studies in this half with the largest code base (FindBugs, with 70833 LOC) and smallest code base (MineRay, with 3478 LOC) have differences that fall in between this range at 210% and 252%, respectively.

In summary, we believe our aggregate score of 92% should not be taken to say anything more precise than "there is a substantial effect". In particular, this score does not allow us to draw a general conclusion that SubjectJ cuts remodularization time in half. In any case, even if we consider only the first four case studies—where performing remodularizations in SubjectJ before Java negatively biased the time differences—we still see a positive 19% trend.

4.3.2 Number of Bugs Introduced

The programmer introduced many more bugs while remodularizing in Java than in SubjectJ[4]: 8 bugs were introduced in Java remodularizations, and only one bug was introduced

[4] We considered bugs to be distinct if they were detected and fixed between different test runs of the application being remodularized.

Code Base	**SubjectJ** *(hours)*	**Java** *(hours)*	**Difference**
SubjectJ remodularization performed first (total hours = 63.4)			
Tetris	3.0	3.5	+17%
TyRuBa	18.0	20.3	+13%
JHotDraw	4.2	4.0	-5%
Chinese Chess	3.8	6.6	+74%
Sum(1)	29.0	34.4	+19%
Java remodularization performed first (total hours = 66.9)			
MineRay	0.7	2.3	+252%
DrawSWF	2.0	5.8	+183%
FindBugs	9.9	30.7	+210%
JChessBoard	3.0	12.5	+325%
Sum(2)	15.6	51.3	+229%
Aggregated results (total hours = 130.3)			
Sum(1)+Sum(2)	**44.6**	**85.7**	**+92%**

Table 4. Overview of time data results from case studies.

in a SubjectJ remodularization. The manifestation of these bugs and how they contributed to remodularization complexity is described in Section 4.4.4.

4.4 Qualitative Results

After performing each case study, we analyzed our notes and the resulting code bases. The main focus of this analysis was to find anecdotal evidence explaining how static-dynamic coupling made performing the same remodularization in Java more complex than in SubjectJ.

Overall, the Java remodularization process seemed cognitively harder. In both cases the programmer needed to explore the code to find sections relative to the concern of interest. However, in SubjectJ the general remodularization process itself was centered almost exclusively around this type of activity. This was facilitated by the use of SubjectJ Java annotations to mark code of interest, and tracking static dependencies using the Checker tool. In contrast, with standard Java, the programmer needed to not just identify code of interest, but also decide on a separation strategy for changing dynamic structure to allow separating the code into its own Java package. The listener infrastructure from our motivating example is one separation strategy. Four additional strategies were used in the case studies.

Separation strategies added complexity to the Java remodularization process. The complexity manifested itself in multiple ways: difficulties deciding on the "right" strategy; complex transformations not well supported by available automated refactoring tools; and the introduction of bugs.

We will begin by discussing different separation strategies used in the case studies. Then we will go into each of the above problems related to using them in more detail, illustrating each problem with anecdotes from our case studies.

```
public class BugInstance {
    private BugProperty propertyListHead;
    private SAVE_BugInstance saveBugInstance;
    ...
    public BugInstance(...) {
        ...
        saveBugInstance = new SAVE_BugInstance(this);
        ...
    }
    public BugProperty getPropertyListHead() {
        return propertyListHead;
    }
}
public class SAVE_BugInstance {
    private BugInstance bugInstance;
    public SAVE_BugInstance(BugInstance bugInstance) {
        this.bugInstance = bugInstance;
    }
    public void writeXML() {
        ...
        BugProperty prop = bugInstance.getPropertyListHead();
        ...
    }
    ...
}
```

Figure 8. "Dual Object" implementation of saving XML bug reports.

4.4.1 Separation Strategies

Over the course of the 8 case studies, changes to dynamic program structure were frequently needed to achieve the desired code separation. The need to change dynamic structure was much more prevalent in the Java remodularizations than in the SubjectJ remodularizations—this was partially reflected in the number of different separation strategies used.

In the SubjectJ remodularizations, dynamic structure changes were only needed when a method contained code that was related to a concern of interest, and code that was not. The "split method" strategy (supported as the "Extract Method" refactoring in Eclipse) was used to extract the code of interest into a separate method.

In the Java remodularizations, the split method strategy was also needed, but four additional strategies were used. One of the strategies involved applying the listener infrastructure, as in the motivating example. We now describe the three other strategies.

One strategy, which we call "dual object", is generally applicable and was used in every case study except the first. It splits objects of a given class into two objects: one object containing the methods and fields we want to separate, and one containing the rest. For example, the code in Figure 8 is the result of applying dual object to separate the `writeXML` method from a class called `BugInstance`. Each `BugInstance` object becomes two objects, with the `SAVE_BugInstance` objects containing the `writeXML()` method. The dual objects have mutual references to each other; constructor code needs to be added to initialize these references. Accesses from one object to the other are accesses to `this` in the original code, and need to be updated.

Another strategy, which we call "static method", was used only in the TyRuBa case study. This strategy converts a group of instance methods from several classes into a single static method. The receiver object becomes one of the parameters of the static method and method dispatch is converted to "if instanceof" tests. This static method can then be moved around relatively freely because static methods have loose static-dynamic coupling. This strategy avoids the complexity of dual object's mutual references, but is less generally applicable—it can only be applied to move methods, not instance fields. It introduces complexity of its own by using "if instanceof" tests and typecasts.

A final strategy, used only in the DrawSWF case study, splits a class into a superclass and a subclass. The subclass contains the field and method declarations that need to be separated, while the remaining field and method declarations stay in the superclass. References are updated to create and use objects of the subclass. This strategy results in weaker encapsulation than dual object—all protected members of the superclass are exposed to the subclass; and multiple constructor calls must be changed to create the subclass instead of the superclass.

4.4.2 Deciding on the "Right" Strategy

When separating source code from classes in Java, it was often difficult to decide what separation strategy to use. In most cases, the programmer decided to apply the "dual object" strategy, because he found it hard to predict whether all code of interest could be captured by other (possibly more elegant, but less general) strategies.

However, there were situations where experimentation with dual object, and careful consideration of strategy benefits and limitations, led to the use of a different separation strategy. For example, in the "TyRuBa" case study, the abstract `QueryEngine.getStoragePath()` method (and every implementation of it) was identified as one of the methods to be separated from the rest of the code. While starting to apply dual object, the programmer realized that addition of code in `SimpleRuleBaseBucket` to initialize its dual object reference would be necessary. This was complicated because this code could only be placed *after* `super()` constructor calls, causing potential problems if the dual object needed to be accessed during the execution of `super()`. To avoid potential problems, the programmer observed that no fields needed to be moved, and decided to use the static method strategy instead. Although static method caused other complications—typecasts and "if instanceof" tests—these complications seemed more predictable.

In a scenario from the "DrawSWF" case study, separation of code into a subclass was chosen over the dual object strategy. While exposing protected members from superclasses to subclasses results in weaker encapsulation, it was for this reason that the programmer chose to use subclassing in this scenario. In particular, the programmer needed to separate some code from a class which extended the library class `DefaultCellEditor`. The code required access to a protected inner class `DefaultCellEditor.EditorDelegate`. The programmer thus needed to extend `DefaultCellEditor` in the separated code. However, applying dual object and directly extending `DefaultCellEditor` would involve many delegations between the duals, simply to override default implementations of methods in both classes. Noticing that the disadvantages of subclassing would be minimal in this scenario—the original class extending `DefaultCellEditor` was instantiated at only one place in the code—the programmer chose instead to subclass this original class, preserving implementations of overriden methods from the original code. In this particular scenario, applying the subclassing strategy avoided complexity that would be introduced by applying dual object.

In contrast to the above scenarios, remodularizations in SubjectJ did not require making these kinds of decisions because only one strategy, split method, was ever used.

4.4.3 Automation of Transformations

The more complex separation strategies used in the Java remodularizations usually involved manual transformations not well-supported by the available automated refactoring tools. In contrast, the only strategy used in SubjectJ remodularizations was the "split method" strategy, which is relatively well-supported by Eclipse. Other program structure changes in SubjectJ remodularizations were static in nature and well-supported by the SubjectJ tools.

We provide an anecdote from the "FindBugs" case study to illustrate the difference in automated transformation support between SubjectJ and Java remodularizations. While remodularizing using Java, the programmer applied the dual object strategy on 38 classes to separate their `writeXML()` method declarations. An example of the resulting code from one of the classes—`BugInstance`—is shown in Figure 8. Lack of automated refactoring support for this strategy required the programmer to manually change a large number of different places in the code in a coordinated fashion.

While remodularizing using SubjectJ, the programmer also decided to separate the `writeXML` method declaration from the same 38 classes. However, manual code modifications were limited to adding `@Subject` annotations to each `writeXML` method. Further code modifications were largely performed by using the Checker and Decomposer tools. While the changes where similar in extent to the Java remodularization, the programmer did not need to spend much effort manually performing changes.

This anecdote illustrates that the difference between the SubjectJ and Java remodularizations is less in the extent of the changes than it is in how well the changes could be supported by reliable automated tools. Although Eclipse has refactorings to move methods and fields, the programmer did not use them here because they were felt to be unreliable. Indeed, he did try to use Eclipse "move" refactoring tools in the first case study, but found they often introduced errors into the code. We later verified that the "move method" tool in Eclipse passes the receiver object as an argument to the method, essentially using the method as a static method. Thus, if the method is used in a polymorphic way, then references to the method are not updated. The "move field" tool appears to never update references. We do not believe these shortcomings are because of a lack of effort on the part of the Eclipse developers, but rather because strong static-dynamic coupling makes moving instance members in Java a complex problem. In comparison, moving declarations between SubjectJ subjects is relatively uncomplicated because it only affects the (static) interfaces between the subjects.

4.4.4 Introduction of Bugs

Table 5 provides a brief overview of the bugs introduced during the case studies. All 9 bugs were introduced because of mistakes made while making manual code changes to apply separation strategies. More specifically, Bugs #1 to #6 were related to applying the dual object strategy. Bugs #7 and #9 were caused by mistakes made when manually extracting method code. Finally, Bug #8 was related to applying the "subclassing" strategy. We now describe two bugs in more detail—Bug #2, introduced during a Java remodularization; and Bug #9, the only bug introduced during a SubjectJ remodularization.

The programmer introduced Bug #2 while applying dual object, to separate "brain" code from the abstract class `Piece` into `BRAIN_Piece`. The bug resulted from forgetting to properly initialize the references from `BRAIN_Piece` objects to their dual objects. A possible cause for this forgetfulness is the added complexity of applying dual object to an abstract class: the `Piece` class does not (and cannot) call a constructor of the abstract `BRAIN_Piece` class, so the programmer was not alerted to the lack of an explicit `BRAIN_Piece` constructor by compiler errors.

Bug #9 was introduced by the programmer when manually extracting method code in a situation that was not supported by the Eclipse "extract method" tools. A careless mistake resulted in the extracted method shown below:

```
@Subject("Text")
@Export("OTHER")
private static DrawObject createObject2_TEXT(int drawing_mode) {
    if (drawing_mode == TEXT) {
        new Text();
    }
    return null;
}
```

The programmer wrote "`new Text();`" when he intended to write "`return new Text();`". The programmer was not alerted by a compilation error because the method still ended with a return statement, resulting in a `NullPointerException`.

4.5 Experiment Conclusions

The case study results reported on here were focussed on the research question "How and to what extent does static-dynamic coupling in a language impact the complexity of remodularizing programs written in that language?". To this end, we compared remodularization complexity in Java versus SubjectJ. The main difference between SubjectJ and Java is that SubjectJ removes what we perceived to be the greatest source of static-dynamic coupling from Java, by allowing individual class members to be moved easily from one subject to another.

To assess and compare complexity, a series of 8 case studies was performed. Each case study consisted of performing a remodularization task once using Java, and once using SubjectJ. Two quantitative indicators of complexity—time taken and number of bugs introduced—both suggest that remodularizing code is significantly more complex in Java than SubjectJ. The qualitative anecdotal analysis provides some insight into the causes of the complexity.

Central to the complexity of Java remodularizations was the use of complex separation strategies, which led to difficulties not encountered in SubjectJ remodularizations. Decisions on which strategy to use were trivial in SubjectJ—only one strategy was used—but were more complex in Java: multiple strategies were used, and, since they involve changes to dynamic structure, they required a more in depth understanding of how the program works. The extent of the changes between SubjectJ and Java remodularizations was similar. However, the more complex separation strategies used in Java remodularizations are difficult to support via refactoring tools, and hence are typically carried out via an ad-hoc and error-prone "copy and paste and fix errors" approach.

5. Future Work

This paper is focussed specifically on examining the impact of static-dynamic coupling in a language on remodularization complexity. The experimental results are thus based on performing given remodularization tasks with well-defined

#	Code Base	Behavior	Cause
Bugs introduced during Java remodularization tasks			
1	JHotDraw	NullPointerException	Dual is referenced before it is created.
2	Chinese Chess	NullPointerException	Reference to dual is not initialized.
3	FindBugs	NullPointerException	"Getter" method returns `null` instead of reference.
4	FindBugs	Program freeze	"Getter" method calls itself infinitely.
5	JChessBoard	NullPointerException	Dual of `JChessBoard` used before it is fully initialized.
6	JChessBoard	NullPointerException	Dual of `History` used before it is fully initialized.
7	DrawSWF	NullPointerException	Manually extracted code from method erroneously sets local variable to `null`.
8	DrawSWF	IllegalArgumentException	Reference to original class instead of new subclass.
Bugs introduced during SubjectJ remodularization tasks			
9	DrawSWF	NullPointerException	Manually extracted code from method erroneously returns `null`.

Table 5. Overview of bugs introduced during case studies (arbitrarily numbered for reference convenience).

outcomes. The paper does not, however, examine the motivations for performing remodularizations, nor the extent to which remodularized code satisfies motivating goals. In particular, issues such as the resulting understandability, maintainability, and evolvability of remodularized code are outside the scope of our central research question. However, an understanding of such broader issues is important for determining how our experimental results are relevant to practical software engineering processes.

In general, it would be useful to have a better understanding of when and how purely static program structure changes are applicable. For example, although our experimental results suggest that reduced static-dynamic coupling decreases complexity of remodularization tasks, dynamic structure changes have benefits as well. Future work could examine the resulting benefits and disadvantages from static versus dynamic separations of code (e.g. for providing runtime pluggability of different modules). Future work could also explore the merits of using remodularization to achieve better incrementality. As mentioned in our illustrative example of Section 2, implementing a listener infrastructure is sometimes useful, depending on the broader goals behind the code restructuring. In such cases, purely static changes to program structure could be used to improve incrementality of a restructuring effort that eventually leads to dynamic structure changes for a listener based solution.

6. Related Work

This paper is about the impact of static-dynamic coupling on remodularization. Since remodularizations are program transformations intent on preserving program behavior, we can regard them as a specific type of refactorings.

In this section we discuss related work in three broad categories. In Section 6.1 we provide a historical perspective on the concept of static-dynamic coupling. Section 6.2 discusses related work on programming systems that provide loose static-dynamic coupling. Finally, Section 6.3 discusses related work on automated refactoring tools.

6.1 Historical Perspective

An interesting point to note, is that as early as 1969, Dijkstra talked about language design in terms of coupling between static and dynamic program structure and its potential impact on the understandability of programs [9]. Dijkstra motivates structured programming, by arguing that it needs to be easy to map a program's textual structure onto the structure of its execution and vice versa. Essentially, Dijkstra is arguing for strong static-dynamic coupling at the intra-procedural level.

Even though our results seem to argue for loose static-dynamic coupling, we do not believe they inherently contradict Dijkstra's view. First, our results concern static-dynamic coupling in the context of code restructuring, leaving questions on understandability open for future work (see 5). Secondly, we are mainly interested in coupling at the coarse-grained modular level, not the level of sub-method granularity.

Arguably, despite Dijkstra's eloquent arguments for strong static-dynamic coupling, we have seen a trend in programming language evolution that moves away from strong coupling between dynamic and static structure at the coarser level. Examples include the introduction of object-oriented inheritance and dynamic method dispatch [12].

6.2 Static-Dynamic Coupling and Languages

SubjectJ was designed to remove what we perceived to be Java's greatest source of static-dynamic coupling. Besides Hyper/J (the inspiration for SubjectJ, as we described in Section 3), there are many other programming systems that can be considered to have loose static-dynamic coupling. In this section we provide an overview and discuss how SubjectJ relates to these other systems.

The main difference between SubjectJ and Java is that it allows splitting declarations that belong to a single class across multiple source files. There are many other object-oriented programming systems that provide similar functionality. For example, C# [19] supports partial classes. Ruby [11] allows class members to be declared outside of a class. Multi-method languages (e.g. [5, 8, 7]) also decouple the declaration of methods from the declaration of classes.

Some programming systems based on mixins [1] provide mixin layers as units of modularity that crosscut classes [29, 3, 20]. MixJuice is based on the concept of difference-based modules [20]. The idea is that a mixin-layer module implements a particular software feature. Each module can extend existing classes or interfaces, or define additional classes and interfaces.

Aspect-oriented programming languages (e.g. [23, 22, 30]) provide inter-type declarations and advice. Intertype declarations make it easy to move whole declarations out of classes and into aspects, providing a kind of static-dynamic coupling at the granularity of member declarations, similar to SubjectJ. Aspect-oriented languages also provide pointcuts and advice, which in some sense extend static-dynamic decoupling to sub-method granularity.

The above technologies differ in mechanism, but, like SubjectJ, they provide flexibility in organizing the coarse static structure of the programs without changing dynamic structure. So this idea is not new. Recall however that SubjectJ in itself was not intended to be a novel contribution. Indeed, to be close to an ideal guinea pig, SubjectJ was *not* designed to add novel features, but rather to be very similar to Java, while removing what we perceived to be Java's greatest source of static-dynamic coupling.

We claim the value of our contribution is not the invention of new language features, but rather that we provided tangible evidence that existing features which reduce static-dynamic coupling may reduce accidental complexity for remodularization. This idea in itself is also not new. In fact, we believe language designers intuitively understand this. For example, the MixJuice authors explicitly mention that moving *code* between MixJuice super-modules and sub-modules retains the semantics of the code. They suggest that this could make some refactorings easier. However, as far as we know we are the first to attempt quantifying this effect with empirical data.

As noted in Section 4.2.3 however, we should be cautious in trying to generalize too much from our experiment; in particular, there is little grounds to draw conclusions about the impact of pointcut advice on remodularization complexity in AspectJ.

6.3 Refactoring Tools

Refactoring tools can provide substantial support for achieving remodularization goals. There is a large amount of work directed towards providing semi-automated (e.g. [33, 34, 15]) and fully-automated [27, 4] refactoring support for complex refactorings. Some combine this with program slicing [35] to aid in the selection of what code to extract and make refactoring tools more precise [14, 24].

However, we believe that the level of support that can be provided by automated refactoring tools is indirectly affected by static-dynamic coupling, because it is dependent on the complexity of program transformations performed by the tools. In other words, refactorings that require more subtle or complex tranformations are less likely to be implemented, or, if they are implemented, more likely to be unreliable. Indeed, we found in our experiment that the difference between the SubjectJ and Java remodularizations was less in the extent of the changes than it was in how well the changes could be reliably supported by available tools. In particular, the Eclipse IDE does not provide a reliable refactoring for moving instance methods between classes. It should be noted that moving static methods does not have the same problem. Similarly, moving methods between SubjectJ subjects is relatively uncomplicated because it only affects the static interfaces between subjects. Thus SubjectJ is able to reliably support this type of transformation through tools that are of comparable complexity to tools for moving Java static methods.

This is not to suggest that developing reliable and sophisticated refactoring tools is not a useful endeavour. In fact, we believe that work on refactoring tools, and work on programming language features that reduce static-dynamic coupling and make code easier to move, are complementary and should go hand in hand. For example, aspect-oriented programming languages which reduce static-dynamic coupling create opportunities to catalogue whole new classes of refactorings for remodularizing code into aspects [26], and automate the refactoring process [10, 15, 16].

7. Conclusion

In this paper we asked the following central research question:

> *How and to what extent does static-dynamic coupling in a language impact the complexity of remodularizing programs written in that language?*

The key contribution of our paper is that we provide experimental results towards answering this research question. We performed a series of remodularization case studies in both Java and SubjectJ. To be close to an ideal guinea pig, SubjectJ was designed explicitly to be very similar to Java, while relaxing what we perceived to be Java's greatest source of static-dynamic coupling: the constraint that a class's declaration must be wholly contained within a single Java source file.

Our results include quantitative data suggesting that the impact of relaxing this restriction is substantial. In particular, analysis of the quantitative results suggests that remodularizing in Java takes considerably more time and results in more frequent introduction of bugs than remodularizing in

SubjectJ. Our results also provide some insight into *how* accidental complexity arises in remodularization of Java programs because of the constraint. Specifically, remodularizing in Java involves more complex separation strategies—strategies for changing dynamic structure solely to allow code separation. The complexity of these strategies is such that they are often poorly supported by available refactoring tools and hence are carried out via an ad-hoc and error-prone "copy and paste and fix compiler errors" approach.

References

[1] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA, 1990. ACM.

[2] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[3] Richard Cardone and Calvin Lin. Comparing frameworks and layered refinement. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 285–294, Washington, DC, USA, 2001. IEEE Computer Society.

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

[5] Craig Chambers. Object-oriented multi-methods in Cecil. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, volume 615 of *LNCS*, pages 33–56, Utrecht, the Netherlands, June 1992. Springer-Verlag.

[6] Rick Chern. Reducing remodularization complexity through modular-objective decoupling (in progress). Master's thesis, The University of British Columbia, 2008.

[7] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd D. Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA-00)*, volume 35.10 of *ACM Sigplan Notices*, pages 130–145, N. Y., October 15–19 2000. ACM Press.

[8] Linda G. Demichiel. Overview: The Common Lisp Object System. *Lisp and Symbolic Computation*, 1(2):227–244, September 1988.

[9] Edsger W. Dijkstra. *Notes on Structured Programming*, chapter 1, pages 1–82. Academic Press, 1972.

[10] Ran Ettinger and Mathieu Verbaere. Untangling: a slice extraction refactoring. In Karl Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 93–101. ACM Press, March 2004.

[11] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly, 2008.

[12] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2005.

[14] William G. Griswold and David Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, July 1993.

[15] Jan Hannemann, Thomas Fritz, and Gail C. Murphy. Refactoring to aspects: an interactive approach. In *eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 74–78, New York, NY, USA, 2003. ACM.

[16] Jan Hannemann, Gail Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In Peri Tarr, editor, *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*, pages 135–146. ACM Press, March 2005.

[17] William H. Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In *OOPSLA*, pages 411–428, 1993.

[18] William H. Harrison, Harold Ossher, and Peri L. Tarr. General composition of software artifacts. In Welf Löwe and Mario Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 194–210. Springer, 2006.

[19] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[20] Yuuji Ichisugi and Akira Tanaka. Difference-based modules: A class independent module mechanism. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, Malaga, Spain, June 2002. Springer Verlag.

[21] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *AOSD*, pages 178–187, 2003.

[22] Kabir Khan, Bill Burke, Flavia Rainone, Staale Pedersen, Marc Fleury, Adrian Brock, Claude Hussenet, and Marshall Culpepper. JBoss AOP. http://labs.jboss.com/jbossaop/.

[23] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Budapest, Hungary, June 2001.

[24] Raghavan Komondoor and Susan Horwitz. Effective, automatic procedure extraction. In *IWPC*, page 33. IEEE Computer Society, 2003.

[25] P. Li and E. Wohlstadter. View-based maintenance for graphical user interfaces. In *Proc. of the International Conference on Aspect-Oriented Software Development*, 2008.

[26] Miguel P. Monteiro and João M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 111–122, New York, NY, USA, 2005. ACM.

[27] Ivan Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *OOPSLA*, pages 235–250, 1996.

[28] Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz, and Vincent Kruskal. Subject-oriented composition rules. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 235–250, New York, NY, USA, 1995. ACM.

[29] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In Eric Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 550–570. Springer, 1998.

[30] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60. Australian Computer Society, Inc., 2002.

[31] P. Tarr and H. Ossher. Hyper/J user and installation manual. Technical report, IBM T. J. Watson Research Center, 2000.

[32] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of ICSE '99*, pages 107–119, Los Angeles CA, USA, 1999.

[33] Frank Tip. Refactoring using type constraints. In Hanne Riis Nielson and Gilberto Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2007.

[34] Lance Tokuda and Don S. Batory. Evolving object-oriented designs with refactorings. In *Proceedings of Automated Software Engineering*, page 174, 1999.

[35] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.