Evolving Object-Oriented Applications with Refactorings

Lance Tokuda and Don Batory
Department of Computer Science
University of Texas at Austin
Austin, TX 78712-1188
{unicron, batory}@cs.utexas.edu

ABSTRACT[1]

Refactorings are behavior-preserving program transformations that automate design level changes in object-oriented applications. Many schema transformations, design patterns, and hot-spot meta-patterns are automatable. Thus, it seems possible to develop practical tools that could significantly simplify the evolution of object-oriented applications by automating common, yet tedious and error-prone, tasks.

Our research evaluates whether refactoring technology can be transferred to the mainstream by restructuring non-trivial C++ applications. The applications that we examine were evolved manually by software engineers. We show that an equivalent evolution could be reproduced significantly faster and cheaper by applying a handful of general-purpose refactorings. In one application, over 14K lines of code were transformed automatically that otherwise would have been coded by hand. Our experiments expose requirements, limitations, and topics of further research before refactoring technology can be delivered to a production environment.

1 Introduction

Before the invention of *graphical user interface (GUI)* editors, the process of evolving a GUI was to design, code, test, evaluate, and redesign again. With the introduction of editors, GUI design has become an interactive process allowing users to design, evaluate, and redesign an interface on-screen and to output compilable source code that reflects the latest design.

We believe that a similar change needs to occur for editing object-oriented class diagrams. Editing a class diagram can be as simple as adding a line between classes to represent an inheritance relationship or moving a variable from a subclass to a superclass. However, such changes must now be accompanied by painstakingly identifying lines of affected source code, manually updating the source, testing the changes, fixing bugs, and retesting the application until the risk of new errors is sufficiently low. Furthermore, designs can require a great deal of experimentation [Joh88]. Multiple iterations of the design-implement-test cycle may be required to achieve a satisfactory final design.

Just as GUI editors revolutionized GUI design, we believe that class diagram editors (where changes to an application's diagram automatically trigger corresponding changes to its underlying source code) will revolutionize the evolution of software design. The technology to power such a tool is *refactorings* — behavior-preserving program transformations that automate[2] many design level[3] changes. Automation significantly reduces, if not eliminates, the burden of identifying and modifying source code to affect design changes. Modifications are done correctly, thereby reducing costly and tedious debugging and testing that would otherwise have to be performed. And it facilitates experimentation: if some design changes are deemed inappropriate, it is a substantially easier task to undo these changes and apply another sequence of refactorings that are more appropriate. If such changes had to be done manually, their difficulty might preclude attempts to make them.

This paper presents the results of applying refactorings to replicate the work of software engineers/designers on a pair of non-trivial C++ applications. Our experiments expose requirements, limitations, and topics of further research before refactoring technology can be delivered to a production environment.

2 Refactorings

A *refactoring* is a parameterized behavior-preserving program transformation that updates an application's design and underlying source code. A refactoring is typically a very simple transformation that has a straightforward (but not necessarily trivial) impact on application source. An example is **inherit[Base, Derived]**, which establishes a superclass-subclass relationship between two classes, **Base** and **Derived**, that were previously unrelated. From the perspective of an object-oriented class diagram, the **inherit** refactoring merely adds an inheritance relationship between the **Base** and **Derived** classes, but also it alters the application's source code to reflect this change.

A refactoring is more precisely defined by (a) a purpose, (b)

---

1. We gratefully acknowledge the sponsorship of Microsoft Research, the Defense Advanced Research Projects Agency (Cooperative Agreement F30602-96-2-0226), and the University of Texas at Austin Applied Research Laboratories.

2. The term *automate* refers to a refactoring's programmed check for enabling conditions and its execution of all source code changes. The choice of which design to implement and which refactorings need to be applied is always made by a human.

3. We use a limited definition of the term *design* referring to the aspect of design reflected in the extended class diagram notation from Gamma [Gam95] (See Section 2).

arguments, (c) a description, (d) enabling conditions, (e) an initial state, and (f) a target state. Such a definition for **inherit[Base, Derived]** is given in Figure 2.1. A summary of the class diagram notation used in this example and throughout this paper is presented in Figure 2.2. For the most part, aspects of a refactoring are self-explanatory. The most complicated aspect deals with a refactoring's enabling conditions.

2.1 Enabling Conditions

Programs are restructured by applying a series of refactorings. Because individual refactorings preserve behavior, a series of refactorings also preserves behavior. To preserve behavior, we adopt the method proposed by Banerjee and Kim for database schema evolutions [Ban87] and employed by Opdyke for refactorings [Opd92]. A set of invariants is defined which, if preserved, guarantees that two programs will run identically. When a refactoring runs the risk of violating an invariant, enabling conditions are added to guarantee that the invariant is preserved.

Opdyke identified seven invariants that preserve the behavior of C++ and Smalltalk programs. For example, his first invariant is that each class must have a unique superclass and its superclass must not also be one of its subclasses. The first enabling condition of the **inherit** refactoring (Figure 2.1) preserves this invariant.

Refactorings have been found to be useful even when predicated on conservative enabling conditions. For example, the **inherit** transformation is conservatively limited to single inheritance systems by Opdyke's first invariant. While support for multiple inheritance systems is possible, it was not necessary for transforming the applications described in this paper or for adding numerous design patterns and hot-spot meta patterns [Tok99].

Most but not all enabling conditions can be verified automatically. In the **inherit** example, the first three conditions are verified automatically while the last enabling condition must be verified by the user to guarantee that behavior will be preserved (see discussion in Section 4.2).

2.2 Design Evolution and Refactorings

Three kinds of object-oriented design evolution are: schema transformations, design pattern microarchitectures, and hot-spots. *Schema transformations* are drawn from object-oriented database schema transformations that perform edits on a class diagram [Ban87]. Examples are renaming a class, adding new instance variables, and moving a method up the class hierarchy. *Design patterns* are recurring sets of relationships between classes, objects, methods, etc. that define preferred solutions to common object-oriented design problems [Gam95]. *Hot-spots* are aspects of a program which are likely to change from application to application [Pre95]. Designs using abstract classes and template methods are prescribed to keep these hot-spots flexible.

Each example of design evolution mentioned above can be implemented by one or more refactorings. The list of refactorings used in our research is given in Table 1. This

Name:
Inherit[Base, Derived]

Purpose:
To establish a superclass-subclass relationship between two existing classes.

Arguments:
Base - superclass name
Derived - subclass name

Description:
Inherit[] makes **Base** a superclass of **Derived**. `vm*()` represents the unimplemented virtual methods inherited by **Base** subclasses.

Enabling Conditions:

- **Base** must not be a subclass of **Derived** and **Derived** must not have a superclass.
- Subclasses of **Base** must support methods `vm*()` if objects of that class are created. Otherwise, there will be no implementations for `vm*()`.
- Initializer lists must not be used to initialize **Derived** objects. Initializer lists must initialize aggregates and aggregates cannot have superclasses [Ell90].
- Program behavior must not depend on the size of **Derived**. Adding a superclass can affect its size. This condition cannot be verified automatically.

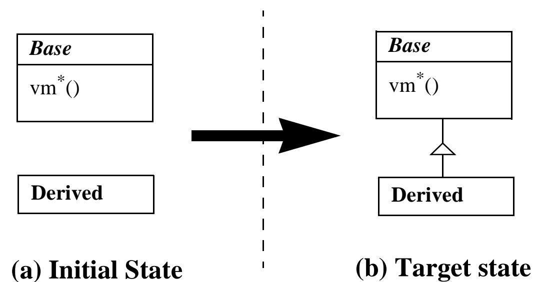


Figure 2.1: Inherit[Base, Derived] transformation

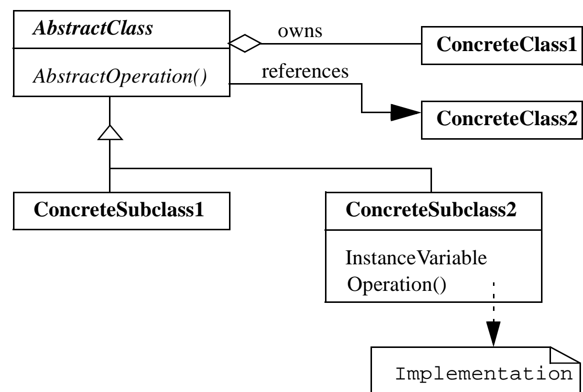


Figure 2.2: Notation

table includes refactorings proposed by Banerjee and Kim for evolving object-oriented database schemas [Ban87] and by Opdyke for restructuring object-oriented programs [Opd92]. We found that transforming actual C++ programs required additional refactorings. We enlarged the set of schema evolutions to include, for example, **inherit** (from the example above) and **substitute**. **Substitute** changes a class's dependency on a class **C1** to a dependency on a superclass of **C1** [Tok95]. Other refactorings are language-specific; **procedure_to_method** and **structure_to_class** convert C artifacts to their C++ equivalents. Yet another set of refactorings supports the addition of design pattern microarchitectures in evolving programs [Tok95, Tok99]. Examples include **add_factory_method**, **singleton**, and **procedure_to_command**. **Add_factory_method** creates a method which returns a new object, **singleton** ensures that a class will have only one instance, and **procedure_to_command** converts a C procedure to a singleton class with a method for executing the procedure. The refactorings that we added to the lists of Banerjee, Kim, and Opdyke are italicized in Table 1.

Schema Refactorings

- add_variable
- create_variable_accessor
- *create_method_accessor*
- rename_variable
- remove_variable
- push_down_variable
- pull_up_variable
- *move_variable_across_object_boundary*
- create_class
- rename_class
- remove_class
- *inherit*
- *uninherit*
- *substitute*
- rename_method
- remove_method
- push_down_method
- pull_up_method
- move_method_across_object_boundary
- extract_code_as_method
- *declare_abstract_method*
- *structure_to_pointer*

C++ Refactorings

- *procedure_to_method*
- *structure_to_class*

Pattern Refactorings

- *add_factory_method*
- *create_iterator*
- *composite*
- *decorator*
- *procedure_to_command*
- *singleton*

Table 1: Object-oriented refactorings

It is worth noting that there have been surprisingly few implementations of refactorings. The first implementations of Opdyke's refactorings were by Tokuda [Tok95], Roberts [Rob97], and Schulz et. al. [Suc98]. To our knowledge, we were first to implement refactorings for design patterns [Tok95] and hot-spot meta patterns [Tok99].

Given that schema transformations, design patterns, and hot-spot meta patterns are used frequently in evolving designs, many of which are automatable as (one or more) refactorings, we expected to replicate some, but not all, design changes in our experiments.

3 Evolving Applications

We selected SEMATECH's CIM Works and CMU's Andrew User Interface System as examples of evolving applications. They were chosen based on availability of source code with a version history, size, and presence of design changes. The following features make this study unique:

Replication of design evolution. Designs were extracted from two versions of the same application. The older design became the initial state and the newer design became the target state. Our objective was to determine if a sequence of refactorings could be applied to transform the initial state to the target state. By doing so, we would be automating changes that were performed manually by the original application designers. This correspondence makes comparison of automation versus hand-coding valid and provides us with a key indicator: how often refactorings could be used.

Our approach is consistent with a principled development style which performs improvements by first transforming the design while preserving program behavior, and then extending the better designed system [OSh86, Cas91].

Non-trivial Applications. Transforming large applications tests refactoring scalability. Ideas that are effective on small applications of fewer than one thousand lines of code may ultimately fail for real world applications whose size can exceed one hundred thousand lines.

Mainstream object-oriented language. C++ was chosen as the target language for experimentation. It is by far the most widespread object-oriented programming language for practical reasons such as backward compatibility with C, portability, availability of third party compilers and tools, legacy system compatibility, and availability of trained personnel. It was expected that C++'s complexity might introduce problems which would not appear for less popular object-oriented languages. A side benefit of this choice is that most claims for C++ can also be made for the increasingly popular Java programming language.

3.1 Evolving CIM Works

Computer Integrated Manufacturing (CIM) Framework is an industry-wide initiative to define a standardized object-oriented framework for writing semiconductor manufacturing execution systems [Ste95]. CIM Works is a Windows application created to demonstrate and test the SEMATECH CIM Framework specification [McG97].

Major design changes in CIM Works occur between Version 2 and Version 4. The Version 2 design shown in Figure 3.1 stores data and its graphical representation in the same object. For example, **CEquipmentManager** contains methods for adding and removing pieces of equipment to be managed as well as methods for building a GUI menu. The Version 4 design shown in Figure 3.2 separates data and graphics into two class hierarchies. This separation gave Version 4 the freedom to create different views of the same data as with the model-view-controller paradigm [Kra88].

Version 2 is approximately 11K lines of code. The transformation between designs is accomplished in nine steps, each of which is realized by applying a sequence of primitive refactorings:

1. Rename the classes of the original hierarchy to the split hierarchy using **rename_class**. (The original classes retain the GUI aspects of objects, whereas their corresponding “split” classes — created in Steps 2 or 7 — encapsulates object data).
2. Create the concrete data classes **Factory**, **Person**, **EquipManager**, etc. using **create_class** (Figure 3.3a).
3. Add `m_objptr` instance variables to the concrete GUI classes using **add_variable**. `m_objptr` is of the corresponding data class type (Figure 3.3b).
4. Move non-GUI instance variables and methods from the GUI classes to the data classes using **move_variable_across_object_boundary** and **move_method_across_object_boundary**. Data is accessed through the `m_objptr` instance variables (Figure 3.4).
5. Create abstract data classes **Resource**, **CompManager**, **MovementResource**, etc. using **create_class.**
6. Establish inheritance relationships between the abstract data classes and the concrete data classes using **inherit** (Figure 3.3c).
7. Move common instance variables and method declarations up the data class hierarchy using **pull_up_variable** and **declare_virtual_method** (Figure 3.5).
8. Change the type of `m_objptr` from a structure to a pointer using **structure_to_pointer**.
9. Declare the reference between GUI objects and data objects in the abstract classes. References to data objects are made abstract (Figure 3.2).[4]

These steps were executed by 81 refactorings, resulting in a total of 486 lines of CIM Works source being modified.

3.2 Evolving the Andrew User Interface System

The *Andrew User Interface System (AUIS)* from CMU is an integrated set of tools that allow users to create, use, and mail documents and applications containing typographically-formatted text and embedded objects [Mor85]. The two versions under study were Version 6.3 written in C and Version 8.0 converted to C++. Version 6.3

4. In this step, the generalization is made that all **CIcon** objects point to a **Resource** object through the `m_objptr` instance variable. This requires that casts to the appropriate data class are made whenever data object instance variables are referenced through GUI objects. For example:

```
CIcPerson *p = new CIcPerson;
p->person_ptr->f_name = "John";
```

is transformed to:

```
CIcPerson p;
((Person *)p->m_objptr)->f_name =
    "John";
```

It is unclear if this was the correct design decision since the GUI classes are specific to a single data class. This step was not automated although it would be possible to do so.

Figure 3.1: Version 2 Design

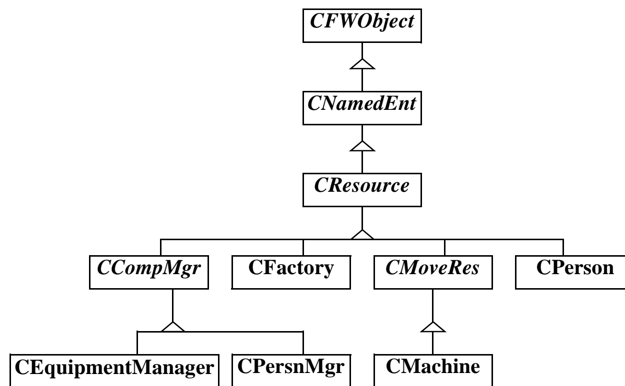


Figure 3.2: Version 4 Design

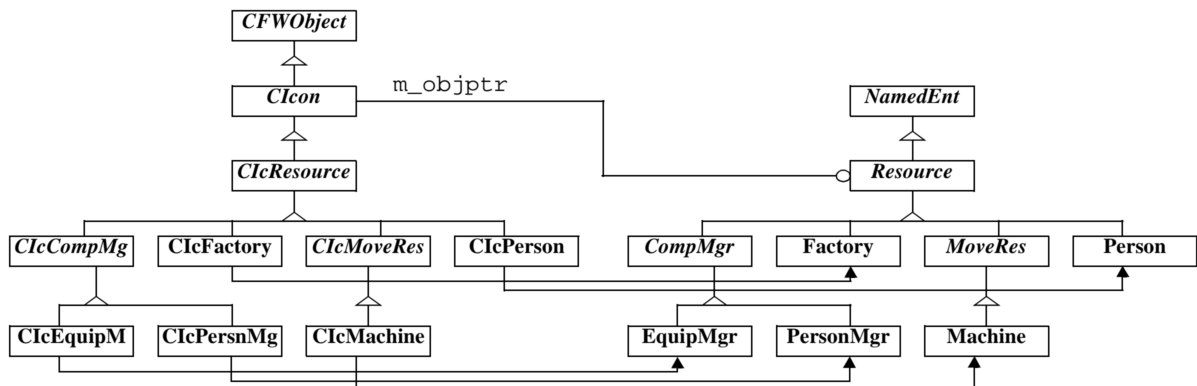


Figure 3.3a: Original classes renamed and Data Classes created

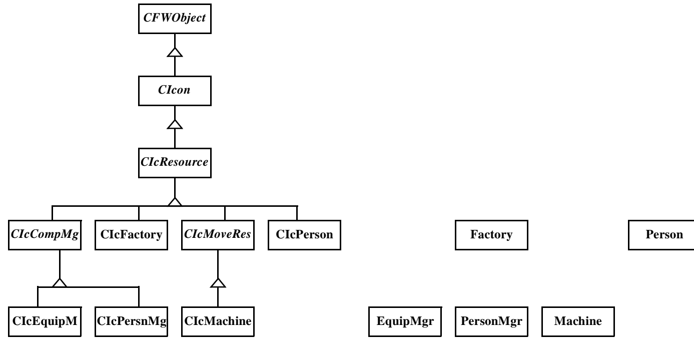


Figure 3.3b: Connect GUI and Data Classes

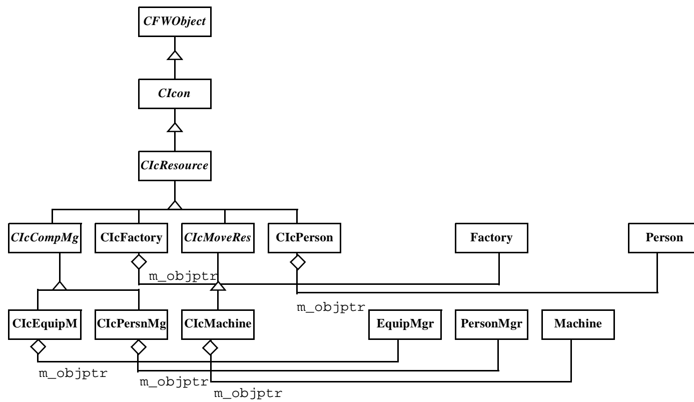
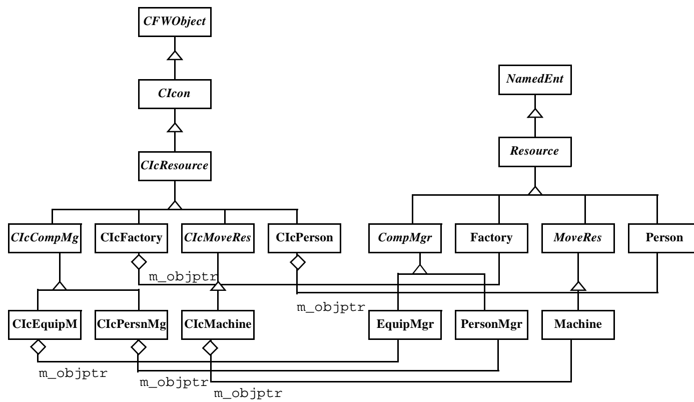


Figure 3.3c: Create Abstract Data Class Hierarchy



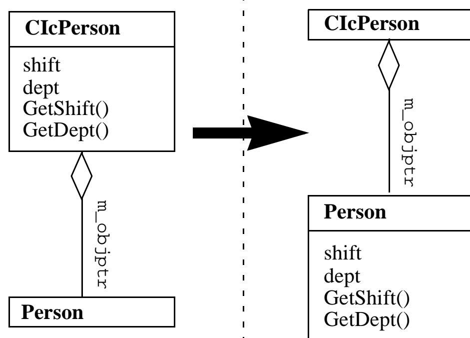


Figure 3.4: Instance variables and methods moved to data classes

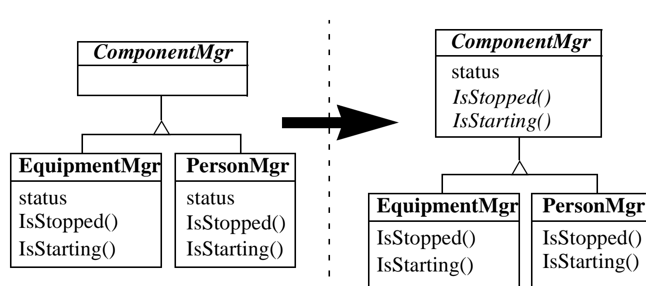


Figure 3.5: Instance variables and method declarations moved to abstract classes

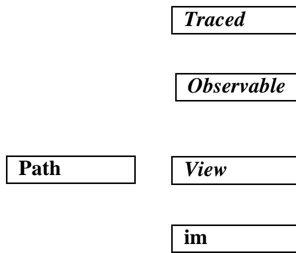


Figure 3.6: Structures converted to classes

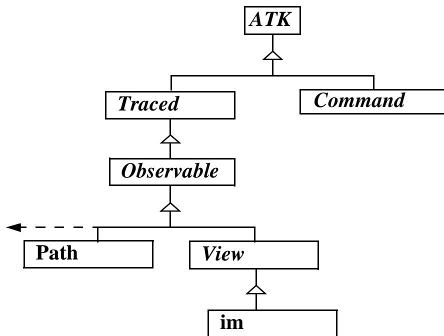


Figure 3.7: Class hierarchy created

stores actions as function pointers while Version 8.0 supports and recommends creation of a separate subclass for each action (similar to the Command design pattern[5]). Over ninety classes using almost 800 actions are affected. The transformation is accomplished in five steps.

1. Convert Version 6.3 C structures to C++ classes using **structure_to_class** (Figure 3.6).
2. Create the **ATK** and **Command** abstract classes using **create_class**.
3. Establish the inheritance relationships between **ATK** and other classes using **inherit** (Figure 3.7).
4. Derive command classes for each action using **procedure_to_command**. Figure 3.8 displays the result of transforming `PlayKbdMacro()` into a **Command** subclass. The newly created **PlayKbdMacroCmd** contains an `Execute()` method which calls `PlayKbd-Macro()`. It also contains an `Instance()` method which returns a unique instance of the class. Using `Instance()` instead of `new` to create objects guarantees that a pointer to a **PlayKbdMacroCmd** object is unique.
5. Convert procedure pointers to commands using **procedure_ptr_to_command**. In this step, the data types for structures using procedure pointers are converted to use **Command** pointers, procedure calls are converted to use `Execute()` methods, and procedure assignments are converted to use `Instance()` methods. Figure 3.9 displays the transformation of the **bind_Description** structure. The `proc` instance variable is converted to a **Command** pointer.

5. The Command design pattern objectifies an action. The action is triggered by calling an `Execute()` method implemented in each derived class [Gam95].

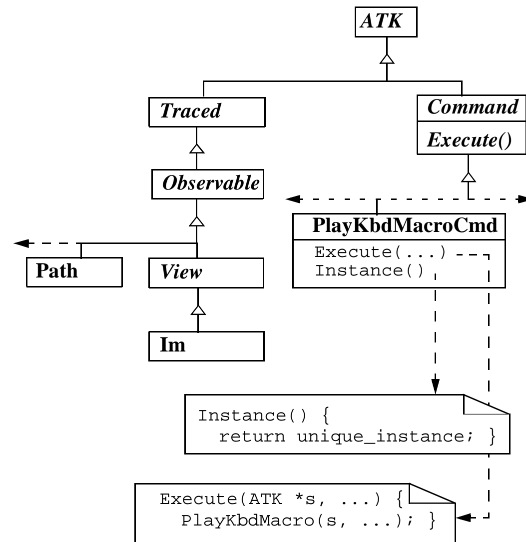


Figure 3.8: Software Microarchitecture for Im and Command classes

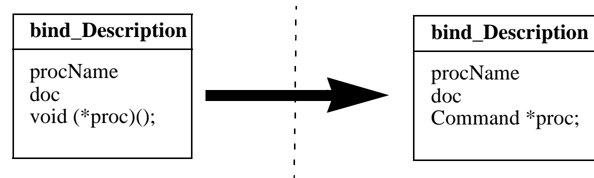


Figure 3.9: Convert procedure pointer to Command pointer

All steps were executed with a total of 800 refactorings. This number is large but could be significantly reduced (see discussion on Granularity of Transformations in Section 4.4).

The conversion of AUIS to use the new **Command** class for its hundreds of actions requires approximately 14K lines of code changes. It is interesting to note that although the action class is supported and recommended for all new changes to AUIS, the existing code base was never migrated to this new mechanism. Thus, in Version 8 there are actually two different representations for actions: the original code used function pointers while all new additions to AUIS use command classes. Our transformed version of AUIS converted all source to use action classes. The volume of changes might explain why Version 8.0 code was never fully converted to use its newly defined action class. Concomitantly, this also suggests an advantage of refactorings to perform large edits automatically, which people might not undertake by hand.

4 Introspection and Lessons Learned

Our experiments provided a tremendous learning experience in evaluating refactoring technologies. In the following sections, we present some of the more important lessons that

we learned on refactoring benefits and limitations, as well as on the requirements that must be satisfied and research problems that must be solved for refactoring technology to succeed.

4.1 Refactoring Benefits

Automating Design Changes. The most important result of our research is to establish that refactorings can automate significant design changes in real world applications. For CIM Works, the main class hierarchy was split into two connected hierarchies in an automated way. For AUIS, procedure pointers were converted to use the command design pattern generating over 14K lines of code changes.

It is of interest to compare the effort required to perform these changes manually versus the effort when aided by refactorings. We estimate that the CIM Works changes would take us two days to implement and debug by hand versus two hours when aided by refactorings. We estimate that the AUIS changes would require two weeks to implement and debug by hand versus one day when aided by refactorings. The number of refactorings and their subsequent cost could be reduced in both experiments with larger grain refactorings to provide a substantially greater benefit (see Granularity of Transformations, Section 4.4).

Earlier we mentioned that many, but not all, design patterns, schema transformations, and hot-spot meta patterns could be expressed as refactorings. In both experiments, it was possible to automate *all* design changes. This was fortuitous. It does suggest, however, that a refactoring tool could service a variety of design changes.

Reduced Testing. In principle, refactorings can reduce testing because they are behavior-preserving. Initially, however, we would expect users to run a full slate of tests on refactored code. The reason is trust: today's compilers aren't verified, yet we have learned to trust their output; we would expect the same for a refactoring tool.

Simpler Designs. Refactorings reduce the need for overly complex designs. Gamma et. al. note that a common design pattern pitfall is over-enthusiasm: "Patterns have costs (indirection, complexity) therefore [one should] design to be as flexible as needed, not as flexible as possible". Designs which attempts to anticipate too many future extensions may also be more error prone with less static type checking[6]. Gamma's example of over-enthusiasm [Gam96] is displayed in Figure 4.1. Instead of creating a simple **Circle** class, an overenthusiastic designer adds a **Circle** factory with strategies for each method, a bridge to a **Circle** implementation, and a **Circle** decorator. The design is likely to be more complex and inefficient than what is actually required. The migration from a single **Circle** class to the complex microarchitecture on the right hand side of Figure 4.1 can be viewed as a transformation. This transformation is, in fact, automatable with refactorings [Tok99]. Thus,

6. Many design patterns use runtime composition versus inheritance as an extension mechanism [Gam95]. The dynamic nature of composition precludes static typechecking.

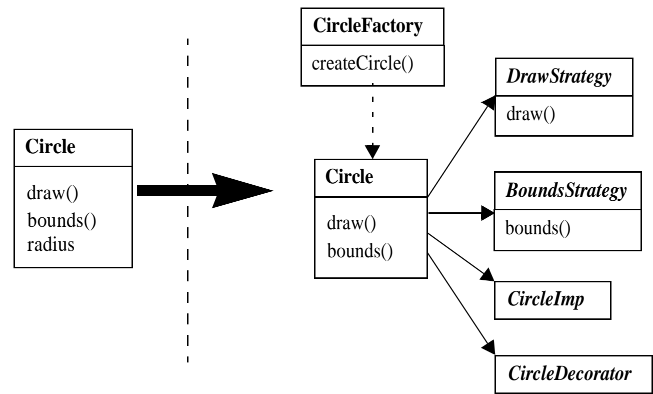


Figure 4.1: Overenthusiastic use of design patterns

instead of over-designing, one can start with a simple **Circle** class and add the Factory Method, Strategy, Bridge, and Decorator design patterns [Gam95] as needed.

Refactorings are capable of extending designs in multiple ways. They encourage designers to create lean designs for the task at hand and to extend those designs with refactorings as new capabilities are needed.

Validation Assistance. The target designs achieved with refactorings were known to be valid, however, this will not be true for most evolving designs. Enabling condition checks can help to establish that a new design is legal or they can point out conflicts between a code level implementation and a desired design change. For example, a programmer may decide to move an instance variable from a base class to a derived class without realizing that objects of the base class type access the instance variable being moved. Enabling condition checks will detect this error. Refactorings are capable of detecting errors resulting from a long series of changes which would be costly to perform and undo manually.

Ease of Exploration. Refactorings allow designers to experiment with new designs. While schema evolutions and design patterns are manually coded into applications today, it is clear that automating their introduction will allow designers to more easily explore a design space without major commitments in coding and debugging time. Ultimately, it may be the ability to evolve and explore new designs that will attract designers to this technology.

4.2 Refactoring Limitations

This section identifies limitations of refactoring systems operating in a mainstream environment. Experiments with large applications revealed limitations which were not issues in previous work on small proof-of-concept programs. While admittedly we anticipated a number of points raised below, we didn't anticipate them all, nor did we realize how significant these points actually were. We discuss our most important observations to alert future researchers to the problems that they will face.

Preprocessor Directives. Our C++ program transformation

tool cannot deal with preprocessor directives because preprocessor directives are not part of the C++ language. The programs in our experiments were preprocessed before being transformed and at that point, preprocessor information could no longer be recovered. In this section we examine the different types of preprocessor information and note workarounds when possible.

#include <filename> — Special comments are inserted to mark the beginning and end of each included file so the files can be unincluded after all refactorings are completed.

#define <constant> n — in some cases these declarations can be replaced by a statement of the form 'const <type> <constant> n' or a list of #define's can be replaced by an enumerated type. Otherwise, this preprocessor information cannot be maintained.

#define macro(x) F(x) — this can sometimes be replaced by an inline function whose return type must be known. For function-like macros which use the # and ## operators, it may not be possible to create an equivalent inline function and the original source code must be changed.

#define FLAG and #ifdef — this information is difficult to maintain. One possible way to retain code that would be removed by #ifdef statements is to store it as a comment which is later uncommented after the source has been refactored. This solution would allow a program to be transformed correctly given one set of compiler flags but it could not guarantee correctness for a different set of flags. This problem is related to the issue of transforming program families discussed in Section 4.4.

We found that while much of the preprocessor information can be dealt with automatically, it is generally not possible to handle all cases that arise in large software applications.[7]

Enabling Conditions. Most but not all enabling conditions can be verified automatically. For the **inherit** refactoring in Figure 2.1, the first three enabling conditions can be checked automatically. For example, it is possible to examine an abstract syntax tree and verify that **Base** is not a subclass of **Derived** (the first enabling condition). Opdyke identifies two enabling conditions which cannot be verified automatically [Opd92]:

- Program behavior must not be dependent on the size of objects.
- Program behavior must not be dependent on the physical layout of objects.

The **inherit** refactoring requires that programs be independent of object size since adding a superclass can change the size of an object. Size and layout were not issues with the two programs transformed in this paper or other programs transformed in [Tok95, Rob97], however, users of refactorings must be aware of this limitation.

7. Recent but unpublished work on Microsoft Research's IP project (to our knowledge) embodies the most advanced attack to date on this problem [Sim98].

4.3 Refactoring Requirements

Source File Access. Enabling conditions are currently written with the assumption that all source code for a program is transformable. This may not be the case. For example, a user may attempt to move an instance variable from a user-defined subclass to a superclass defined in a proprietary third party framework. This was not an issue in this research because the design changes being replicated with refactorings were limited to developer code. For a transformation system targeting a mainstream development environment, checks must be made to determine if any files affected by a transformation are read-only.

Refactorings are currently unable to adequately deal with preprocessor information defined in read-only header files. The constants defined in system, Microsoft Windows, or X Windows header files cannot be converted to variables as suggested in Section 4.2 because the header files are read-only. Replacing the constants with actual values in source code risks poor code readability and incompatibility with future versions of the header files.

Makefiles. Refactorings intended for use in mainstream C++ development environments should accept a makefile as an argument. Makefiles define the set of files for a target application and specify what compilation flags are set. Makefile compatibility minimizes the cost of integrating refactorings into a development environment.

Preserving Comments. A requirement for any source to source transformation systems is that source code comments must be preserved. In the Andrew example, comments accounted for more than 20% of all non-blank lines. The Sage++ toolkit used in this research demonstrates that it is possible to preserve comments while refactoring C++ programs [Bod94]. Comments can also be automatically inserted to document changes resulting from refactorings.

One difficulty is determining which comments apply to a body of source code. For example, at the beginning of a file, one might find comments describing the purpose of the file followed by comments describing the implementation of a method followed by the source code for the method. If the method is moved to another class located in another file, there is no way to distinguish between the comments which are specific to the method and those which describe the entire file.

Code Placement. Issues may arise about where generated code should be placed. Code placement is currently done automatically, however, it may be preferable to give the user options in a production system. For example, a default behavior may be to create a new header and source file whenever a new class is created but the user may prefer to define the class in the same file as some other existing class. In the Andrew example, we would have defined the command subclasses in the file containing the action executed by the command. Adding a new file to a program implies knowledge about the file structure and makefiles.

4.4 Future Research

Our work focused on the practicality of applying primitive refactorings to evolving object-oriented applications. Beyond implementation of required functionality, we identify four issues which require further research.

Granularity of Transformations. The refactorings developed for this research were intended to be primitive and composable to perform more complex refactorings. We did not attempt to minimize the number of refactorings required. In the CIM Works example, the number of refactorings was large (81) although the conceptual number of transformation steps was small (8). One way to reduce the number of refactorings would be to provide larger grain refactorings. In the CIM Works example, the number of refactorings would be significantly reduced if refactorings to move multiple variables and methods were available. Similarly for the Andrew example, most of the 800 transformations take place in Step 4 — converting action procedures to **Command** subclasses. A larger grain transformation which converted a list of procedures to **Command**'s could execute Step 4 in a single transformation. This would reduce the total number of transformations to fewer than twenty. It is important to note that the near term goal of our research has been to develop a basis set of primitive refactorings. Larger grain refactorings up to the size of design patterns may be more convenient in practice.

Large grain refactorings can also simplify the check for enabling conditions. It is sometimes easier to verify enabling conditions for a large grain refactoring instead of verifying enabling conditions for an equivalent series of primitive refactorings [Rob97].

Program Families. Parnas argued that software developers should design each program as a member of a family of programs [Par79]. Transformation systems must recognize that many files may be included by multiple programs. When transforming a file used by more than one program, it is desirable for the transformations system to check enabling conditions for all programs which use that file. Otherwise, a file might be transformed safely for one program while causing another program which uses the same file to break. As a simple example, consider the case where two applications `Driver.C` and `Racer.C` link with `Vehicle.C` supporting class **Car**. Suppose we refactor `Driver.C` and `Vehicle.C` to **rename Car** to **Boat**. `Racer.C` will no longer compile because it depends on the **Car** class which no longer exists.

A refactoring supporting program families would need to accept a list of makefile targets for which the transformation must be valid. The situation is further complicated for C++ by conditional compilation flags which imply that different preprocessed versions of a single file should be considered when checking if a transformation can be performed safely.

Integration with Other Tools. Refactorings packaged as individual executables which take a makefile target as an argument are not dependent on the presence of other tools. In this form, refactorings can be integrated into most mainstream development environments because most environments support command-line access to source code.

Higher levels of integration are still possible. We envision integration with an object-oriented modeling tool such as Rational Rose™ which would allow many refactorings to be invoked as operations on a UML diagram. Integration with a source code control system could allow appropriate files to be checked out, transformed, and checked back in with comments describing the refactorings. Attempts to transform protected files would block the refactoring and notify the user. Integration with an IDE such as Microsoft Visual C++™ would allow transformed code and updated makefiles to be displayed immediately in open windows.

Tool Support. There is a definite lack of tools for manipulating application source that are integrated with standard development tools. Further, the performance of program transformations must be fast so the results of a change can be seen immediately through the browser. An efficient representation for handling source-to-source transformations is discussed in [Gri91]. Simonyi presents IP as a general-purpose system for implementing program transformations [Sim95]. Roberts implements efficient refactorings for Smalltalk [Rob97] and Baxter implements transformations for Cobol [Bax97].

4.5 Implications for Java

Java inherits all of C++'s refactoring benefits while avoiding many of its limitations. First, it has no preprocessor which removes a major barrier to a successful C++ implementation. Second, it does not use makefiles which simplifies the process of piecing together the source files to be transformed. Third, code placement is simplified since methods are stored in a file belonging to the class. Java has no free-floating procedures as with hybrid object-oriented languages such as C++. For these reasons coupled with its growing popularity as an internet language, we believe that Java is the best vehicle for transferring refactoring technology to the mainstream.[8] Tools are now being developed to aid in this process [Sim95, Bax97, Bat98].

5 Related Work

Our work distinguishes between specification-to-source and source-to-source transformations. *Specification-to-source transformations* transform high level declarative specifications to compilable source. Examples are compilers or transformation systems for domain-specific languages [Rea86, Bax90, Bat94]. The ratio between lines of specification to lines of code generated can be one-to-ten or higher. In contrast, *source-to-source transformations* transform a program coded in a given language to another program in the same language. They are generally domain-independent and the transformations can be written to support standard programming languages.

Griswold developed source-to-source transformations for

8. When we began our work, tool support and availability of large Java files were nonexistent. This is no longer true today.

structured programs written in Scheme [Gri91]. The goal of this system was to assist in the restructuring of functionally decomposed software. Software designs developed using the classic structured software design methodology [You79] are difficult to restructure because nodes of the structure chart which define the program pass both data and control information. The presence of control information makes it difficult to relocate subtrees of the structure chart. As a result, most transformations are limited to the level of a function or a block of code.

Object-oriented software designs offer greater possibilities for restructuring. Bergstein defined a small set of object-preserving class transformations which can be applied to class diagrams [Ber91]. Lieberherr implemented these transformations in the Demeter object-oriented software environment [Lie91]. Example transformations are deleting useless subclasses and moving instance variables between a superclass and a subclass.

Opdyke coined the term *refactoring* to describe a behavior-preserving program transformation for restructuring object-oriented software systems. Refactorings were inspired by the schema evolutions of Banerjee and Kim [Ban87], the design principles of Johnson and Foote [Joh88] and the design history of the UIUC Choices operating system [May89]. An example application of refactorings is the creation of an abstract superclass [Opd93]. Hursch and Seiter presented a subset of Opdyke's refactorings which operated under an alternative framework for preserving behavior [Hur96]. Refactorings are implemented for C++ [Tok95, Sch98, Tok99] and for Smalltalk [Rob97]. Roberts offers Smalltalk-specific design criteria for a program transformation tool [Rob97]. One criteria which also applies to C++ software is that users should be allowed to name new entities introduced through transformations.

Tokuda and Batory proposed additional refactorings to support design patterns as targets states for software restructuring efforts [Tok95, Tok99]. Refactorings are shown to support the addition of design patterns to object-oriented software systems [Tok95, Rob97, Sch98, Tok99]. Winsen used refactorings to make existing design patterns more explicit [Win96]. Refactorings also support the addition of Pree's [Pre94] hot-spot meta patterns [Tok99].

A number of tools instantiate a design pattern and insert it into existing source code [Bud96, Kim96, Mei97]. Instantiations are not necessarily behavior-preserving, so testing of changes may be required. Meijers checks invariants governing a pattern and repairs violations when possible. Refactorings do not have this pattern-level knowledge.

6 Conclusion

Evolution of an application's design is inevitable and is done manually at great effort and expense. Refactorings are behavior-preserving program transformations that provide a powerful technology by which significant parts of an application's design evolution can be automated.

The ultimate goal of our research is to provide a mainstream tool that makes editing class diagrams as easy as editing user interfaces with a GUI editor. This paper has taken three important steps towards this goal:

- First, we implemented a set of refactorings that can automate a suite of schema transformations, design patterns, and hot-spot meta patterns. They can reduce or eliminate the need to identify lines of affected source, to execute changes manually, and to test those changes.
- Second, we showed that refactorings can scale and be useful on large, real-world applications. We were able to automate thousands of lines of changes with a general-purpose set of refactorings.
- Third, while our experiments clearly showed the benefits that could result from a refactoring tool, they also revealed the requirements, limitiations, and research problems that remain to be addressed before refactoring technology can be transitioned beyond academic prototypes.

Given the success of our experiments and the difficulty in managing C++ preprocessor information, Java should be the next target language, as we believe that it holds the greatest promise for transferring refactoring technology to the mainstream.

References

[Ban87] J. Banerjee and W. Kim. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proceedings of the ACM SIGMOD Conference*, 1987.

[Bat94] D. Batory et.al. Scalable Software Library. In Proceedings of ACM SIGSOFT, December 1993.

[Bat98] D. Batory et. al. JTS: Tools for Implementing Domain-Specific Languages. In *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.

[Bax90] I. Baxter. Design Maintenance Systems. In Communications of the ACM 35(4), April, 1992.

[Bax97] I. Baxter. and C. Pidgeon. Software Change Through Design Maintenance. In *Proceedings of the International Conference on Software Maintenance '97*, IEEE Press, 1997.

[Ber91] P. Berstein. Object-preserving class transformations. In *Proceedings of OOPSLA '91*, 1991.

[Bod94] F. Bodin. Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In *Proc. 2nd Object-Oriented Numerics Conference*, Sunriver, Oregon 1994.

[Bud96] F. J. Budinsky et.al., Automatic code generation from design patterns. In *IBM Systems Journal*, Volume 35, No. 2, 1996.

[Cas91] Eduardo Casais. *Managing Evolution in Object-Oriented Environments: An Algorithmic Approach*. Ph.D. thesis, University of Geneva. 1991.

[Coad 92] P. Coad. Object-Oriented Patterns. In *Communications of the ACM*, V35 N9, pages 152-159,

September 1992.

[Ell90] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1990.

[Gam93] E. Gamma et. al. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings, ECOOP '93*, pages 406-421, Springer-Verlag, 1993.

[Gam95] E. Gamma et.al. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[Gri91] W. Griswold. *Program Restructuring as an Aid to Software Maintenance*. Ph.D. thesis. University of Washington. August 1991.

[Hun95] H. Huni, R. Johnson and R. Engel. A Framework for Network Protocol Software. In *Proceedings of OOPSLA '95*, 1995.

[Hur96] W. Hursch and L. Seiter. Automating the Evolution of Object-Oriented Systems. *International Symposium on Object Technologies for Advanced Software*. Springer-Verlag, March 1996.

[Joh88] R. Johnson and B. Foote. Designing Reusable Classes. In *Journal of Object-Oriented Programming*, pages 22-35, June/July 1988.

[Johnson 92] R. Johnson. Documenting Frameworks with Patterns. In *OOPSLA '92 Proceedings*, SIGPLAN Notices, 27(10), pages 63-76, Vancouver BC, October 1992.

[Kim96] J. Kim and K. Benner. An Experience Using Design Patterns: Lessons Learned and Tool Support, *Theory and Practice of Object Systems*, Volume 2, No. 1, pages 61-74, 1996.

[Kra88] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. In *Journal of Object-Oriented Programming*, pages 26-49, August 1988.

[Lie91] K. Lieberherr, W. Hursch, and C. Xiao. *Object-extending class transformations.* Technical report, College of Computer Science, Northeastern University, 360 Huntington Ave., Boston, Massachusetts, 1991.

[Flo97] G. Florijn, M. Meijers, and P. van Winsen. Tool Support for Object-Oriented Patterns. In ECOOP '97, number 1241 in Lecture Notes in Computer Science, pages 472-495, Springer-Verlag, 1997.

[May89] P. Maydany et.al. A Class Hierarchy for Building Stream-Oriented File Systems. In *Proceedings of ECOOP '89*, Nottingham, UK, July 1989.

[McG97] P. McGuire. Lessons learned in the C++ reference development of the SEMATECH computer-integrated manufacturing (CIM) applications framework. In *SPIE Proceedings*, Volume 2913, pages 326-344, 1997.

[Mor85] J. H. Morris et.al. Andrew: A Distributed Personal Computing Environment. *Communications of the ACM*, March, 1986.

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[Opd93] W. F. Opdyke and R. E. Johnson. Creating abstract superclasses by refactoring. In *ACM 1993 Computer Science Conference*. February 1993.

[OSh86] Tim O'Shea, Kent Beck, Dan Halbert, and Kurt J. Schmucker. Panel on: The learnability of object-oriented programming systems. In *Proceedings of OOPSLA '86*. pages 502-504. November 1986.

[Par79] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(2):128-138, March 1979.

[Pre94] W. Pree. Meta Patterns — A Means for Capturing the Essentials of Reusable Object-Oriented Design. In *Proceedings, ECOOP '94*, Springer-Verlag, 1994.

[Rea86] Reasoning Systems. REFINE User's Guide, Reasoning Systems Inc., Palo Alto, 1986.

[Rob97] D. Roberts, J. Brant, R. Johnson. A Refactoring Tool for Smalltalk. In *Theory and Practice of Object Systems*, Vol. 3 Number 4, 1997.

[Sch98] B. Schulz et. al. On the Computer Aided Introduction of Design Patterns into Object-Oriented Systems. In *Proceedings of the 27th TOOLS Conference*, IEEE CS Press, 1998.

[Sim95] C. Simonyi, "The Death of Computer Languages, the Birth of Intentional Programming", Microsoft Corporation, Sept 1995.

[Tok95] L. Tokuda and D. Batory. Automated Software Evolution via Design Pattern Transformations. In *Proc. 3rd International Symposium on Applied Corporate Computing*, Monterrey, Mexico, October 1995.

[Tok98] L. Tokuda and D. Batory. Automating Three Modes of Object-Oriented Software Evolution. To appear in *COOTS '99*.

[Win96] Pieter van Winsen. *(Re)engineering with Object-Oriented Design Patterns*. Master's Thesis, Utrecht University, INF-SCR-96-43, November, 1996.

[You79] E. Yourdon and L. Constantine. *Structured Design*. Prentice Hall, 1979.