Automated Software Engineering, 8, 89–120, 2001

Evolving Object-Oriented Designs with Refactorings

LANCE TOKUDA unicom@cs.utexas.edu
DON BATORY batory@cs.utexas.edu
Department of Computer Sciences, University of Texas at Austin

Abstract. *Refactorings* are behavior-preserving program transformations that automate design evolution in object-oriented applications. Three kinds of design evolution are: schema transformations, design pattern microarchitectures, and the hot-spot-driven-approach. This research shows that all three are automatable with refactorings. A comprehensive list of refactorings for design evolution is provided and an analysis of supported schema transformations, design patterns, and hot-spot meta patterns is presented. Further, we evaluate 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 identify benefits, limitations, and topics of further research related to the transfer of refactoring technology to a production environment.

Keywords: refactorings, oo design patterns, program transformations, refactoring tools

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 advance 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.

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 many design[1] level changes.

We use the term *automate* to refer 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 to apply is *not* automated and is always made by a person. That is, a person creates the initial application design and manually selects the sequence of refactorings to transform the design. Not all changes to a program can be automated by refactorings. Fixing bugs and improving algorithms are common changes made to applications, but these changes are *not* refactorings. *Refactorings only address automatable modifications to the class structure of an application.* The benefit of refactorings is that they reduce the cost and tedium of debugging and testing commonly performed modifications that would otherwise have to be performed manually.

2. Refactorings

A refactoring is a parameterized behavior-preserving program transformation that automatically updates an application's design and underlying source code. A refactoring is typically a very simple transformation, one that has a straightforward (but not necessarily trivial) impact on application source code. 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, **inherit** merely adds an inheritance relationship between the ***Base*** and **Derived** classes, but it also checks enabling conditions to determine if the change can be made safely and it alters the application's source code to reflect this change. A refactoring is more precisely defined by (a) a purpose, (b) 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 1. Applying refactorings is superior to hand-coding similar changes because it allows an architect to evolve the design of an existing body of code at the level of a class diagram leaving the code-level details to automation. A summary of the class diagram notation used throughout this paper is presented in figure 2.

Ideally, the behavior preservation of refactorings should be proven formally. In practice and in previous research, this generally has not been done for two reasons. First, to automate refactorings requires a significant engineering effort to build compilers that allow abstract syntax tree manipulations of programs and that have convenient metaprogramming facilities for code generation. Such compilers are especially daunting to implement for C++—the language that we have chosen to use in our experiments. Second, proving that refactorings are behavior preserving requires a formal semantics for the target language to be defined. To do this for a language as complicated as C++ is of dubious value, for the semantics will be limited to a particular version of a particular compiler (where subsequent hacks may inadvertently alter the language semantics).

Instead of formal proofs, we adopt the approach originally proposed by Banerjee and Kim for database schema evolutions (Banerjee and Kim, 1987). They identified a set of invariants that preserved the behavior of object-oriented database schemas. Opdyke proposed a similar set of seven invariants to preserve behavior for refactorings (Opdyke, 1992). Opdyke's refactorings were accompanied by proofs which demonstrated that the enabling conditions he identified for each refactoring preserved the invariants. Opdyke did not prove that preserving these invariants preserved program behavior.

As an example, Opdyke's first invariant is that each class must have a unique superclass and its superclass must not also be one of its subclasses. When a refactoring runs the risk of violating an invariant, enabling conditions are added to guarantee that the invariant is

Name:

Inherit[*Base*, Derived]

Purpose:

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

Arguments:

Base - superclass name

Derived - subclass name

Description:

Inherit[] makes ***Base*** a superclass of **Derived**.

1. ***Base*** must not be a subclass of **Derived** and **Derived** must not have a superclass.
2. Member variables of **Derived** must have distinct names from member variables of ***Base*** and its superclasses.
3. A member function of **Derived** which overrides a function must have the same type signature as the function it overrides.
4. Subclasses of ***Base*** must implement any pure virtual methods if objects of that class are created.
5. Initializer lists must not be used to initialize **Derived** objects.
6. For all inherited instance variables whose type is a class, the constructors for those classes cannot have any side-effects outside of object initialization if **Derived** is instantiated.
7. Program behavior must not depend on the size or layout of **Derived**.

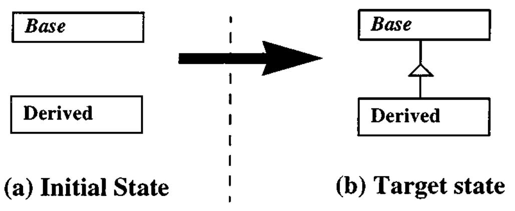

Figure 1. Inherit [*Base*, Derived] transformation.

preserved. Enabling conditions for the **inherit** refactoring are listed in figure 1. **Inherit's** first enabling condition preserves Opdyke's first invariant.

Thus, the engineering approach that we and others have used is to define for each refactoring a set of enabling conditions that are necessary for behavior preservation. Because of the complexity of the languages studied, these conditions may *not* be sufficient. In fact, our experiments show Opdyke's invariants are insufficient for C++. In Section 5.2 we identify counter examples—i.e., additional conditions—that must be considered for behavior preservation. By the same token, there may be additional conditions that even we have not discovered. Until main-stream languages become simple enough for language semantics to

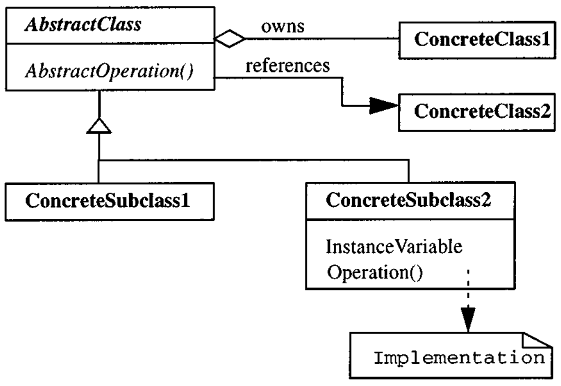

Figure 2. Notation.

be defined formally and for proofs of behavior preservation to be practical, the approach that we have taken is likely to dominate.

There is yet one more difficulty related to enabling conditions: some conditions cannot be checked automatically. For example, condition 7 of figure 1 says that "Program behavior must not depend on the size or layout of **Derived**". That is, if a program exploits knowledge of how **Derived** objects are laid out in memory (e.g., field X has a byte off-set of 5 from the beginning of an object), there is no obvious way to automate the detection of this usage. As another example, it is not possible to verify the "logical correctness" of a refactoring. For example, relating two arbitrary classes via **Inherit** is probably meaningless (e.g., making class **Dog** a subclass of **Building**); but it is possible to perform this refactoring *correctly* even though it makes no sense. In this paper and in any refactoring tool, the logical correctness of refactorings is assumed. In the cases where enabling conditions cannot be checked automatically, a refactoring tool must ask for human intervention to manually verify these conditions (or assert that the conditions are satisfied by the design) before proceeding.

The list of automatable refactorings used in our research is presented in Table 1. Those that are listed in non-italicized font were proposed by Bannerjee, Kim, and Opdyke; refactorings that we have added are *italicized.* In the following sections, we illustrate many of the refactorings in Table 1. For details on refactorings (e.g., code changes, enabling conditions, etc.), we refer readers to Bannerjee and Kim (1987), Opdyke (1992), Tokuda and Batory (1999a,b,c).

3. Automatable modes of design evolution

3.1. Schema transformations

The database schema for an *object-oriented database management system* (*OODBMS*) looks like a class diagram of an object-oriented application. Thus, OODBMS schema transformations have parallels in object-oriented software evolution. An example schema

Table 1. Object-oriented refactorings.

Schema Refactorings	move_method_across_
add_variable	object_boundary
create_variable_accessor	delegate_method_across_object_boundary
create_method_accessor	
rename_variable	extract_code_as_method
remove_variable	*declare_abstract_method*
push_down_variable	*structure_to_pointer*
pull_up_variable	
*move_variable_across_*	C++ Refactorings
object_boundary	*procedure_to_method*
create_class	*structure_to_class*
rename_class	
remove_class	Pattern Refactorings
inherit	*add_factory_method*
uninherit	*create_iterator*
substitute	*composite*
rename_method	*decorator*
remove_method	*procedure_to_command*
push_down_method	*procedure_ptr_to_command*
pull_up_method	*singleton*

transformation is moving an instance variable up the inheritance hierarchy (see figure 3). This transformation is supported by the refactoring **pull_up_variable** which moves an instance variable to a superclass. Banerjee and Kim describe 19 object-oriented database schema transformations of which we implement 12 as automated refactorings.[2] These transformations are listed in Table 2.

Four other useful schema transformations not listed in (Banerjee and Kim, 1987) are given in Table 3. The **move** and **delegate** refactorings introduce a level of indirection when accessing moved methods and variables. Their difference is slight: **delegate** leaves the

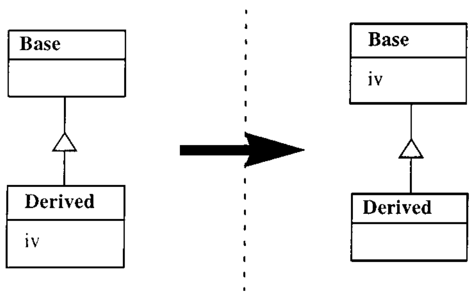

Figure 3. Using pull_up_variable to move instance variable "iv" from derived to base.

Table 2. Banerjee and Kim refactorings.

Description from Banerjee and Kim (1987)	Refactoring from Table 1
Adding a new instance variable	**add_variable**
Drop an existing instance variable	**remove_variable**
Change the name of an instance variable	**rename_variable**
Change the domain of an instance variable	**pull_up_variable** and **push_down_variable**
Drop the composite link property of an instance variable[a]	**structure_to_pointer**
Drop an existing method	**remove_method**
Change the name of a method	**rename_method**
Make a class **S** a superclass of class **C**	**inherit**
Remove class **S** as a superclass of class **C**	**uninherit**
Add a new class	**create_class**
Drop an existing class	**remove_class**
Change the name of a class	**rename_class**

[a]A class **A** with an instance variable of class **B** having the composite link property specifies that **A** owns **B**. **B** cannot be created independently of **A** and **B** cannot be accessed through a composite link of another object.

Table 3. Addition schema refactorings.

Description	Refactoring from Table 1
Move a variable through a composite link	**move_variable_across_object_boundary (figure 4)**
Move a method through a composite link	**move_method_across_object_boundary**
Move a method via delegation through a composite link	**delegate_method_across_object_boundary**
Change a class's dependency on a class **C** to a dependencey on a superclass **S** of **C**	**substitute (figure 5)**

interface of the original class intact by introducing a delegate method. The **move** refactoring eliminates the method from the original class altogether and replaces direct calls to the designated method `foo()` with an indirect call `var->foo()` where variable `var` identifies the class/object to which `foo()` was moved.

The substitute refactoring generalizes a relationship by replacing a subclass reference to that of its superclass (figure 5). This refactoring must be highly constrained, because it does not always work. For example, clients of a subclass can invoke subclass methods that are not present in the superclass. The only time that **substitute** is permitted is under the specific circumstance that the superclass is intentionally designed to represent *the* interface to all of its subclasses; there are no subclass-specific methods or variables that clients should access. This interface might contain only pure virtual functions (which means that all subclasses must provide implementations for these functions) or it might contain methods or method templates that are shared by all subclasses. We discuss a use for **substitute** later in Section 3.2.2 and Section 3.3.1.

Schema transformations perform many of the simple edits encountered when evolving class diagrams. They can be used alone or in combination to evolve object-oriented designs.

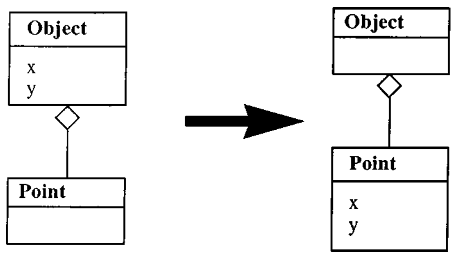

Figure 4. Using move_variable_across_object_boundary to move instance variables x and y.

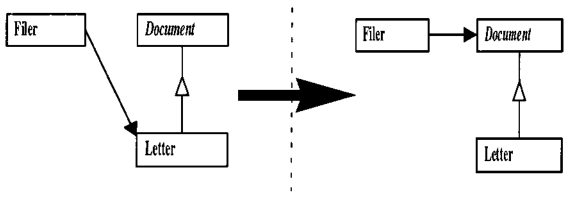

Figure 5. Using substitute to change Filer's reference to a Letter to a reference of a document.

3.2. Design pattern microarchitectures

Design patterns capture expert solutions to many common object-oriented design problems: creation of compatible components, adapting a class to a different interface, subclassing versus subtyping, isolating third party interfaces, etc. Patterns have been discovered in a wide variety of applications and toolkits including Smalltalk Collections (Goldberg, 1984), ET++ (Weinand et al., 1988), MacApp [App89], and InterViews (Linton, 1992). As with database schema transformations, refactorings have been shown to directly implement certain design patterns:

Pattern	Description	Example
Command	Command encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. The **Procedure_to_command** refactorings converts a procedure to a command class.	(Tokuda and Batory, 1999)
Factory Method	Factory Method defines an interface for creating an object, but lets subclasses decide which class to instantiate. **The add_factory_method** refactoring adds a factory method to a class.	(Tokuda and Batory, 1995)
Singleton	Singleton ensures a class will have only one instance and provides a global point of access to it. The **singlton** refactoring converts an empty class into a singleton.	(Tokuda, 1999)

We directly support three additional patterns as refactorings:

Pattern	Description
Composite	Composite composes objects into tree structures to represent part-shole hier-archies. **The composite** refactoring converts a class into a composite class.
Decorator	Decorator attaches additional responsibilities to an object dynamically. The **decorator** refactoring converts a class into a decorator class.
Interator	Iterator provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. The **create_iterator** refactoring generates an iterator class.

While design patterns are useful when included in an initial software design, they are often applied in the maintenance phase of the software lifecycle (Gamma et al., 1993). For example, the original designer may have been unaware of a pattern or additional system requirements may arise that require unanticipated flexibility. Alternatively, patterns may lead to extra levels of indirection and complexity inappropriate for the first software release. A number of patterns can be viewed as automatable program transformations applied to an evolving design. Examples for the following two patterns have been documented:

Pattern	Description	Example
Abstract Factory	Abstract Factory provides an interface for creating families or related or dependent objects without specifying their concrete class.	(Tokuda and Batory, 1995)
Visitor	Visitor lets you define a new operation without changing the classes of the elements on which it operates.	(Roberts et al., 1997)

At least five additional patterns from Gamma et al. (1995) can be viewed as a program transformations:

Pattern	Description
Adapter	Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
Bridge	Bridge decouples an abstraction from its implementation so that the two can vary independently.
Builder	Builder separates the construction of a complex object from its representation so that the same construction process can create different representations.
Strategy	Strategy lets algorithms vary independendently from the clients that use them.
Template Method	Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure.

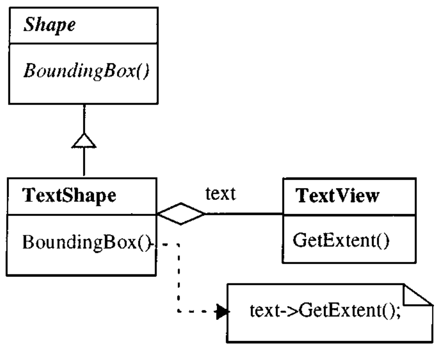

Figure 6. TextShape adapts TextView's interface.

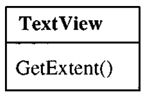

Figure 7. Unadapted text View class.

In all cases, we can apply refactorings to simple designs to create the designs used as prototypical examples in Gamma et al. (1995). The following sections show how the first two patterns can be automated.

3.2.1. Adapter. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. In the object adapter example from Gamma et al. (1995) (figure 6), the **TextShape** class adapts **TextView's** `GetExtent()` method to implement `Bounding-Box()`. The adapter can be constructed from the original **TextView** class (figure 7) in five steps:

1. Create the classes **TextShape** and ***Shape*** using **create_class**.
2. Make **TextShape** a subclass of ***Shape*** using **inherit** (figure 8).
3. Add the `text` instance variable to **TextShape** using **add_variable** (figure 9).

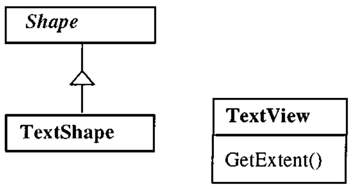

Figure 8. Adaptor class created.

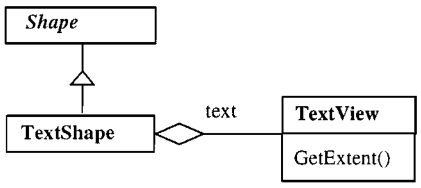

Figure 9. Adaptee instance variable to adaptor.

4. Create the `BoundingBox()` method which calls `text->GetExtent()` using **create_method_accessor**. Not only is the method created, **create_method_accessor** also replaces expressions `text->GetExtent()` with the call to `BoundingBox()`.
5. Declare `BoundingBox()` in ***Shape*** using **declare_abstract_method** (figure 6).

3.2.2. Bridge. Bridge decouples an abstraction from its implementation so that the two can vary independently. In the example from Gamma et al. (1995) (figure 10), the ***Window*** abstraction and ***WindowImp*** implementation are placed in separate hierarchies. All operations on ***Window*** subclasses are implemented in terms of abstract operations from the ***WindowImp*** interface. Only the ***WindowImp*** hierarchy needs to be extended to support another windowing system. We refer to the relationship between ***Window*** and ***WindowImp*** as a bridge because it bridges the abstraction and its implementation, allowing them to vary independently.

Refactorings can be used to install a bridge design pattern given a simple design committed to a single window system. Figure 11 depicts a system designed for X-Windows.

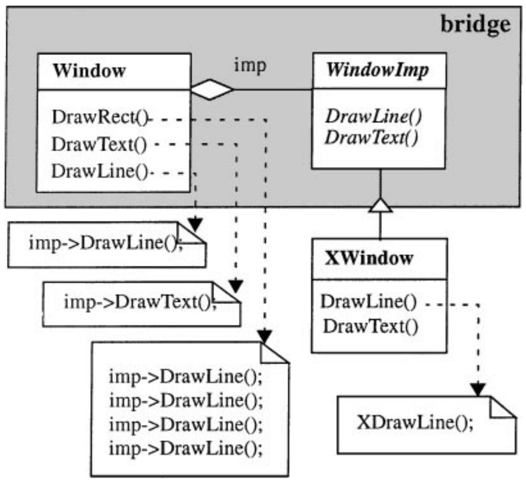

Figure 10. Bridge design pattern example.

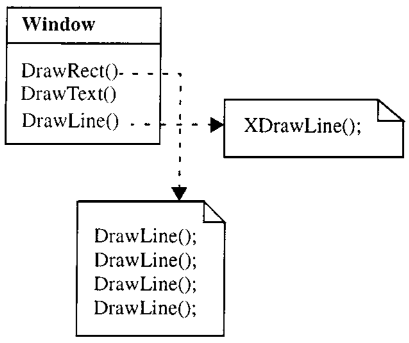

Figure 11. Design for a single window system.

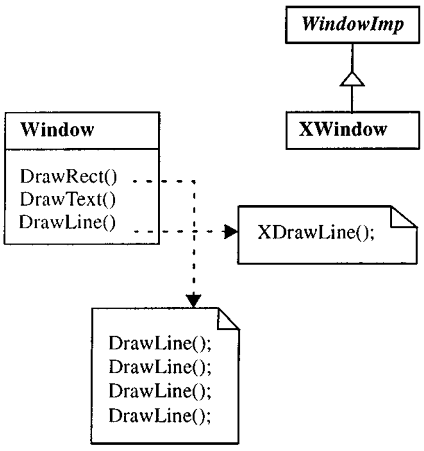

Figure 12. Implementor classes created.

This system can be evolved with refactorings to use the bridge design pattern in six steps:

1. Create classes **XWindow** and ***WindowImp*** using **create_class**.
2. Make ***WindowImp*** a superclass of **XWindow** with **inherit** (figure 12).
3. Add instance variable `imp` to the Window class using **add_variable** (figure 13).
4. Move methods `DrawLine()` and `DrawText()` to the **XWindow** class using the refactoring **delegate_method_across_object_boundary.** This refactoring moves the bodies of the `DrawLine()` and `DrawText()` methods to **XWindow**, while leaving a delegate method in the **Window** class (i.e., the **XWindow** methods are accessed through the `imp` instance variable (figure 14)).
5. Declare method `DrawLine()` and `DrawText()` in ***WindowImp*** with **declare_abstract_method**.

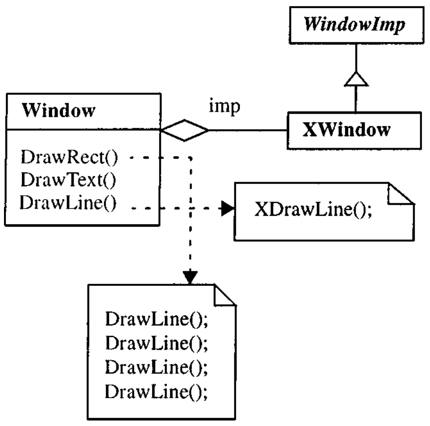

Figure 13. Implementor instance variable added to Window.

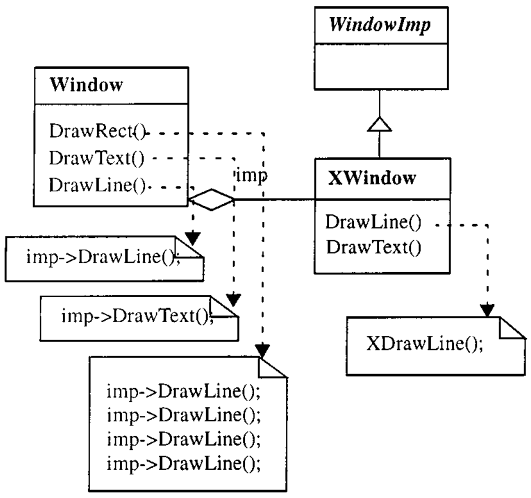

Figure 14. Window system-specific methods moved to XWindow class.

6. Change the type of instance variable `imp` from **XWindow** to ***WindowImp*** using **substitute** (figure 10). This is possible because ***WindowImp*** defines the interface that all its subclasses, such as **XWindow**, must implement.

The Bridge design pattern uses object composition to provide needed flexibility. Object composition is also present in the Builder and Strategy design patterns. The trade-offs between use of inheritance and object composition are discussed in Gamma et al. (1995, pp. 18–20). Refactorings allow a designer to safely migrate from statically checkable designs using inheritance to dynamically defined designs using object-composition.

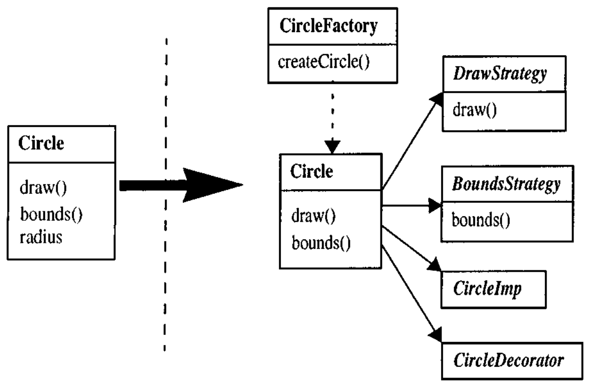

Figure 15. Overenthusiastic use of design patterns.

3.2.3. Role of refactorings for design patterns. Gamma et al. note that a common design pattern pitfall is overenthusiasm: “Patterns have costs (indirection, complexity) therefore [one should] design to be as flexible as needed, not as flexible as possible.” The example from (Gamma et al, 1996) is displayed in figure 15. Instead of creating a simple **Circle** class, an over-enthusiastic 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 in figure 15 can be viewed as a transformation. This transformation is in fact automatable with refactorings.[3] Thus, instead of overdesigning, one can start with a simple **Circle** class and add the Factory Method, Strategy, Bridge, and Decorator design patterns as needed. Refactorings can restructure existing implementations to make them more flexible, dynamic, and reusable, however, their ability to affect algorithms is limited. Patterns such as Chain of Responsibility and Memento require that algorithms be designed with knowledge about the patterns employed. These patterns are thus considered fundamental to a software design because there is no refactoring enabled evolutionary path which leads to their use. Refactorings allow a designer to focus on fundamental patterns when creating a new software design. Patterns supported through refactorings can be added on an if-needed basis to the current or future design at minimal cost.

3.3. Hot-spot analysis

The *hot-spot-driven-approach* (Pree, 1994) identifies which aspects of a framework are likely to differ from application to application. These aspects are called *hot-spots.* When a data hot-spot is identified, abstract classes are introduced. When a functional hot-spot is identified, extra methods and classes are introduced.

3.3.1. Data hot-spots. When instance variables between applications are likely to differ, Pree prescribed the creation of abstract classes. Refactorings have repeatedly demonstrated

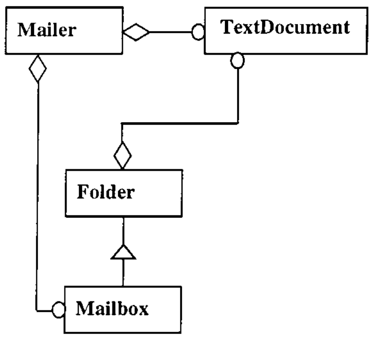

Figure 16. Initial state of mailing system.

the ability to create abstract classes (Opdyke and Johnson, 1993; Tokuda and Batory, 1995; Roberts et al., 1997). As an example, Pree and Sikora provide a Mailing System case study (Pree and Sikora, 1995). Figure 16 displays the initial state of its software design. In this system, **Folder** cannot be nested, and only **TextDocument** can be mailed. Their suggested design is displayed in figure 17. Under the improved design, **Folders** can be nested and any subclass of ***DesktopItem*** can be mailed. Refactorings can automate most of these changes:

1. Create a ***DesktopItem*** class using **create_class** (figure 18). Create an abstract method in ***DesktopItem*** for every public method in **TextDocument** using **declare_abstract_method**; ***DesktopItem*** is to be the interface to all objects that can be mailed.
2. Make ***DesktopItem*** a superclass of **TextDocument** using **inherit** (figure 19).
3. Generalize the link between **Mailer** and **TextDocument** to a link between **Mailer** and ***DesktopItem*** using **substitute** (figure 20). Subclasses of ***DesktopItem*** can now be mailed.

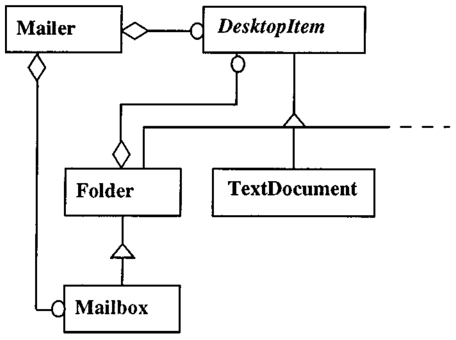

Figure 17. Final state of mailing system.

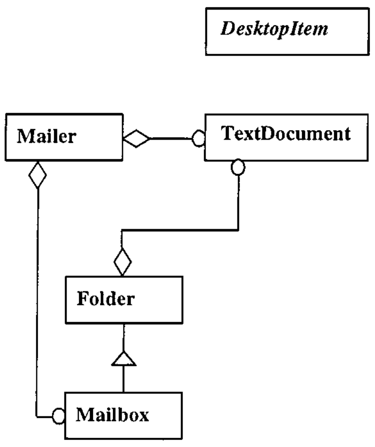

Figure 18. Empty TextDocument class created.

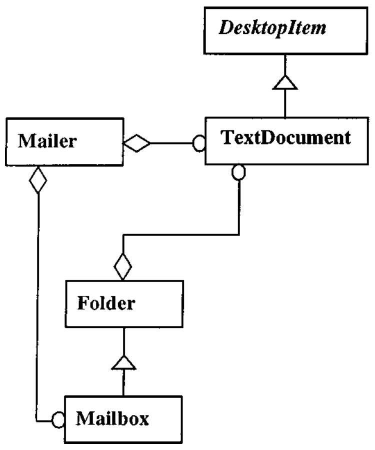

Figure 19. TextDocument inherits from *DesktopItem*.

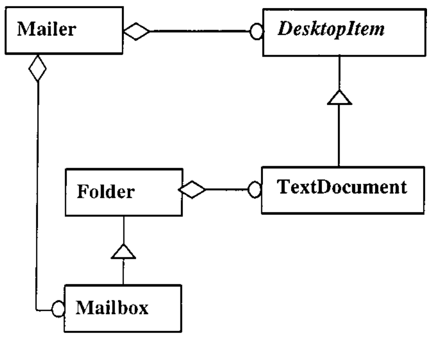

Figure 20. Mailer dependency changed from TextDocument to *DesktopItem*.

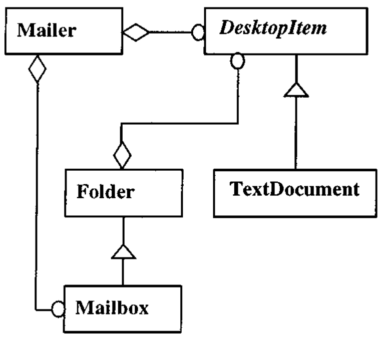

Figure 21. Folder can contain any *DesktopItem.*

4. Generalize the link between **Folder** and **TextDocument** to a link between **Folder** and ***DesktopItem*** using substitute (figure 21). **Folder** can now contain any ***DesktopItem***.
5. This step cannot be automated: add methods to **Folder** that implement the abstract methods of ***DesktopItem***. Such methods would include the procedures for mailing **Folder** objects, and would allow **Folder** objects to be treated as ***DesktopItem*** objects.
6. Make **Folder** a subclass of ***DesktopItem*** using **inherit** (figure 17). A **Folder** which can contain a ***DesktopItem*** can now contain another **Folder**, and can now be mailed like any other ***DesktopItem.***

With the improved design, ***DesktopItem*** provides a superclass for adding other types of media to be mailed. While not all changes to the original design are automatable, most are.

3.3.2. Functional hot-spots. For the case of differing functionality, solutions based on template and hook methods are prescribed to provide the needed behavior. A *template method* provides the skeleton for a behavior. A *hook method* is called by the template method and can be tailored to provide different behaviors. Figure 22 is an example of a template method and hook method defined in the same class. Different subclasses of **T** can override hook method M2() which leads to differing functionality in template method M1() (figure 23). Pree identifies seven meta patterns for template and hook methods: unification,

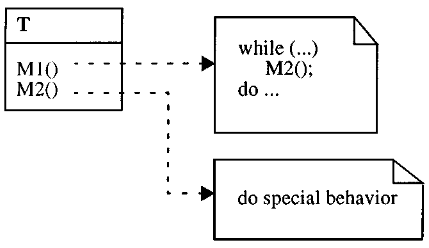

Figure 22. Template and hook methods in same class.

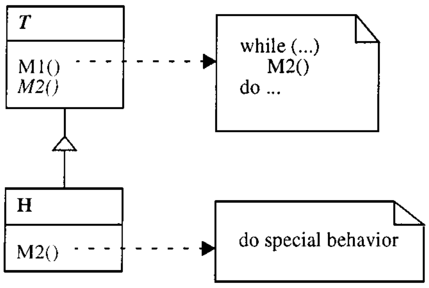

Figure 23. Hook method M2() overridden in class H.

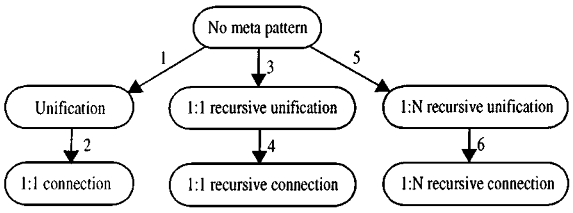

Figure 24. Hot-spot meta pattern transitions enabled by refactorings.

1 : 1 connection, 1 : N connection, 1 : 1 recursive connection, 1 : N recursive connection, 1 : 1 recursive unification, and 1 : N recursive unification (Pree, 1994). Refactorings automate the introduction of meta patterns into evolving designs. The transitions between patterns enabled by refactorings are displayed in figure 24.[4] As examples, we demonstrate how the first two transitions might be expressed using refactorings.

In the unification composition, both the template and hook methods are located in the same class (figure 22). The behavior of the template is changed by overriding the hook method in a subclass (figure 23). A design with no template or hook methods can be transformed to use the unification meta pattern (transition 1 in figure 24). To see how this is possible, consider the class diagram in figure 25 with class **T** having method M1() which calls some special behavior. A hook method can be added with refactorings in one step:

1. Create a hook method M2() which executes the special behavior using **extract_code_as_method** (figure 22). **Extract_code_as_method** replaces a block of code with a call to a newly created method which executes the block.

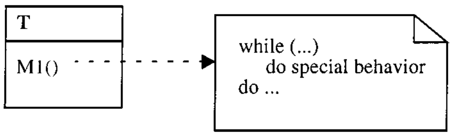

Figure 25. Method M1() calls a special behavior which differs for each application.

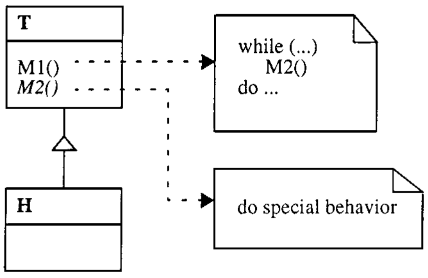

Figure 26. Hook class created.

In the new microarchitecture, general behavior is contained in template method `M1()` while special behavior is captured by hook method `M2()`. To extend the design, subclasses of **T** override `M2()` to provide alternative behaviors for `M1()`. The extended structure can be added in three steps:

1. Create class **H** using **create_class**.
2. Make **T** a superclass of **H** using inherit (figure 26).
3. Move the implementation of `M2()` into **H** using **push_down_method** (figure 23). This refactoring preserves the original interface of **T**, but introduces an abstract method `M2()` that is overridden by **H**.

As a second example, we outline the transition from unification to 1 : 1 connection (transition 2 in figure 24). Consider the 1 : 1 connection meta pattern which stores the hook method in an object owned by the template class (figure 27). Behavior can be changed at run-time by assigning a hook object with a different behavior to the template class. 1 : 1 connection can be automated in three steps using the unification pattern (figure 22) as a starting point.

1. Create class **H** using **create_class**.
2. Add an instance variable `ref` of type **H** to **T** with **add_variable** (figure 28).
3. Move `M2()` to class **H** using **move_method_across_object_boundary** (figure 27).

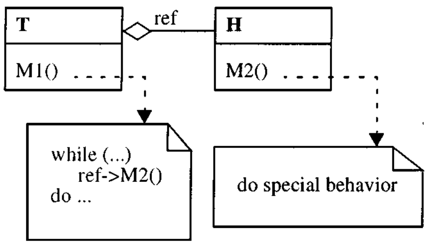

Figure 27. 1 : 1 connection.

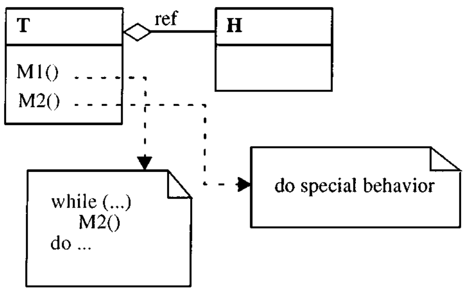

Figure 28. Connection to H object created.

The behavior of template method `M1()` can now be altered dynamically by pointing to different hook class objects with different implementations of `M2()`. Other transitions in figure 24 have similar descriptions.

3.3.3. Role of refactorings in hot-spot analysis. The hot-spot-driven-approach provides a comprehensive method for evolving designs to manage change in both data and functionality. Pree notes that “the seven composition meta patterns repeatedly occur in frameworks.” Thus, we expect an ongoing need to add meta patterns to evolving designs. The addition of meta patterns is currently a manual process. Conditions are checked to ensure that a pattern can be added safely, lines of affected source code are identified, changes are coded, the system is tested to check for errors, any errors are fixed and the system is retested. Retesting continues until the expected likelihood of an error is sufficiently low.

This section suggests that most meta patterns can be viewed as transformations from a simpler design. Refactorings automate the transition between designs granting designers the freedom to create simple frameworks and add patterns as needed when hot-spots are identified.

4. 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 our 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 automate 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.

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 in 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.

4.1. 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 (Stewart, 1995). CIM works is a Windows application created to demonstrate and test the SEMATECH CIM Framework specification (McGuire, 1997).

Major design changes in CIM Works occur between Version 2 and Version 4. The Version 2 design shown in figure 29 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 30 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 (Krasner and Pope, 1988).

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:

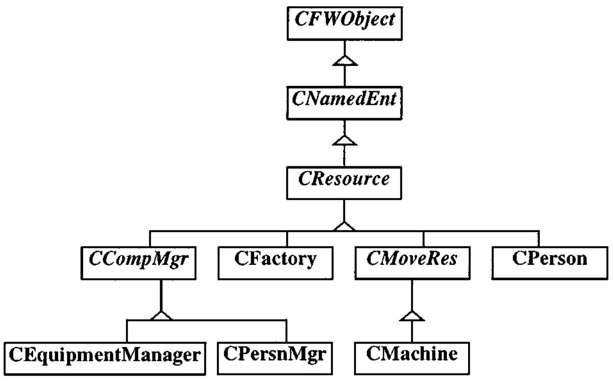

Figure 29. Version 2 design.

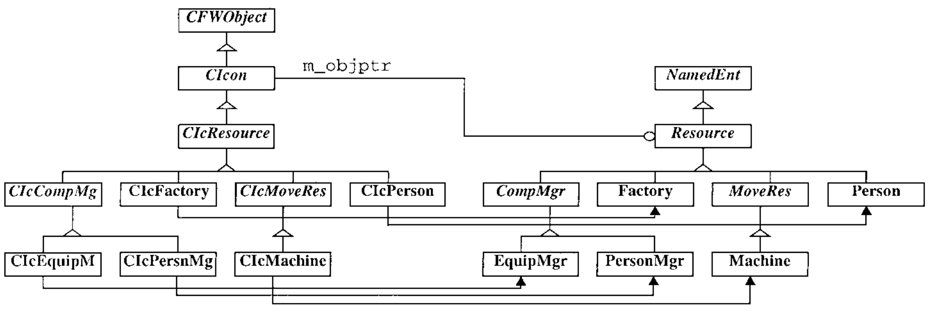

Figure 30. Version 4 design.

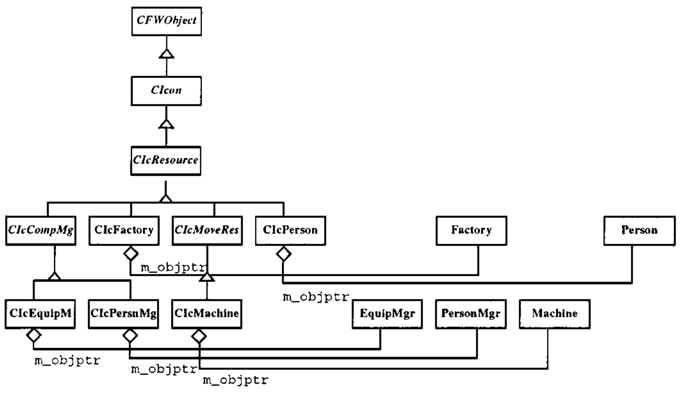

Figure 31. Connect GUI and data classes.

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 and 4—encapsulate object data).
2. Create the concrete data classes **Factory**, **Person**, **Equip-Manager**, etc. using **create_class.**
3. Add `m_objptr` instance variables to the concrete GUI classes using **add_variable**. `m_objptr` is of the corresponding data class type (figure 31).
4. Create abstract data classes **Resource**, **CompManager**, **MovementResource**, etc. using **create_class.**
5. Establish inheritance relationships between the abstract data classes and the concrete data classes using **inherit** (figure 32).
6. 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 33).
7. Move common instance variables and method declarations up the data class hierachy using **pull_up_variable** and **declare_abstract_method** (figure 34).
8. Change the type of `m_objptr` from a structure to a pointer using **structure_to_pointer.**

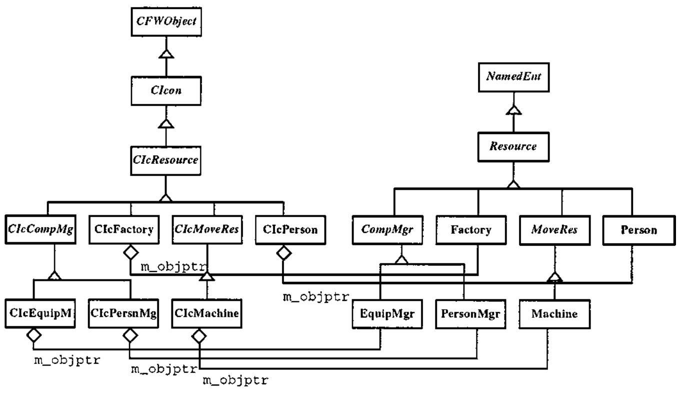

Figure 32. Create abstract data class hierarchy.

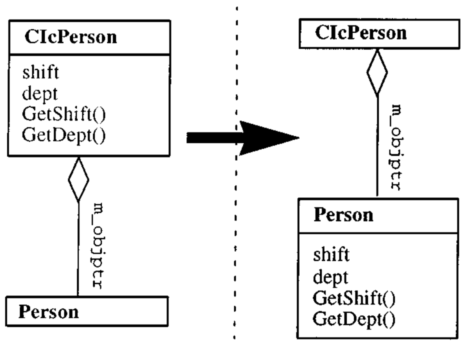

Figure 33. Instance variables and methods moved to data classes.

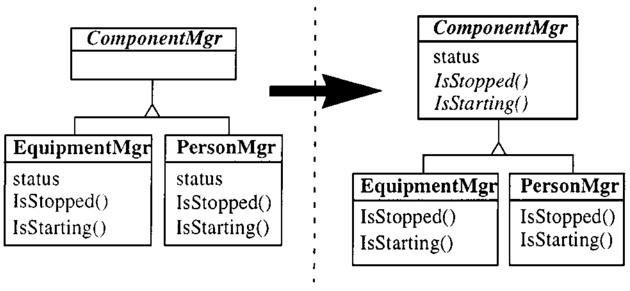

Figure 34. Instance variables and method declarations moved to abstract classes.

9. Declare the reference between GUI objects and data objects in the abstract classes. References to data objects are made abstract (figure 30).[5]

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

4.2. 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 (Morris et al., 1986). The two versions under study were Version 6.3 written in C and Version 8.0 converted to C++. Version 6.3 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).[6] 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 35).
2. Create the **ATK** and **Command** abstract classes using **create_class**.
3. Establish the inheritance relationships between **ATK** and other classes using **inherit**.
4. Derive command classes for each action using **procedure_to_command**. Figure 36 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 37 displays the transformation of the **bind_Description** structure. The `proc` instance variable is converted to a **Command** pointer.

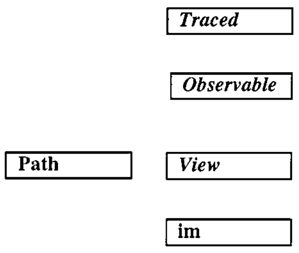

Figure 35. Structures converted to classes.

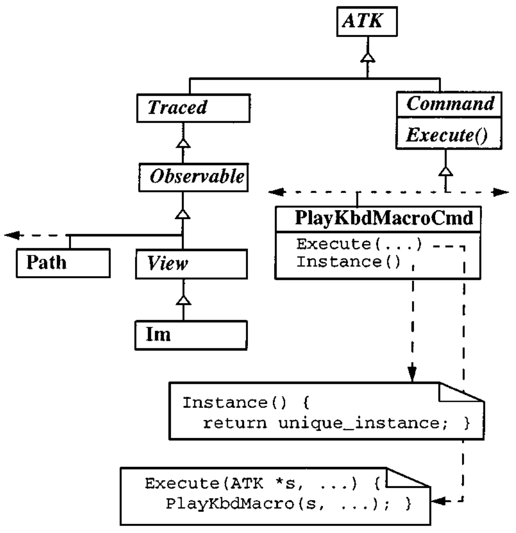

Figure 36. Software microarchitecture for Im and Command classes.

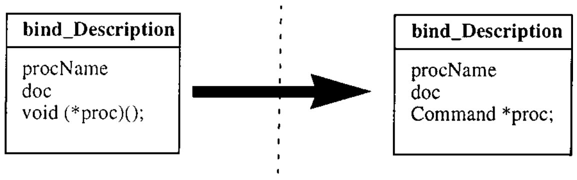

Figure 37. Convert procedure pointer to Command pointer.

All steps were executed with approximately 800 refactorings resulting in 14K lines of code changes.[7]

5. 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, limitations, and research problems that must be solved for refactoring technology to succeed. Further discussion is given in Tokuda and Batary (1999).

5.1. Refactoring benefits

Automating design changes. The most important result of our research is to establish that refactorings can automate significant design changes involving thousands of lines of code in real world application. 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.*

Reduced testing. A good refactoring implementation can reduce the effort required to test new designs. When refactorings preserve behavior, only hand coded changes need to be tested.

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 attempt to anticipate too many future extensions may also be more error prone with less static type checking.[8] 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. Enabling condition checks can detect 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 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 explore more easily a design space without major commitments in coding and debugging time.

5.2. Refactoring limitations

Experiments with large applications revealed limitations which were not issues in previous work on small proof-of-concept programs. 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. While we believe that workarounds are possible for the majority of the cases, it is generally not possible to handle all problems that arise in large software applications.

Conservative enabling conditions. 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 (Tokuda and Batory, 1999).

Automated verification of enabling conditions. Most but not all enabling conditions can be verified automatically. For the **inherit** refactoring, the first five conditions can be checked automatically (figure 1).

Opdyke identifies a condition which cannot be verified automatically: program behavior must not be dependent on the size or layout of objects (Opdyke, 1992). Size and layout were not issues with the two programs transformed in this paper or other programs transformed in (ToKuda and Batory, 1995; Rober et al., 1997), however, users of refactorings must be aware of this limitation.

Behavior preservation. Our experiments revealed that preservation of Opdyke's invariants was not sufficient to guarantee preservation of behavior. Three additional invariants required for C++ are:

1. **Implementation of pure-virtual functions.** If a class is concrete (and thus can be instantiated), it can not have pure-virtual functions.
2. **Maintaining aggregate objects.** If a program depends on the aggregate property of an object, then that property must be preserved.[9]
3. **No instantiation side-effects.** If a refactoring can change the frequency or order in which classes are instantiated, then the constructor cannot have any side-effects beyond initializing the object created.

In light of this discovery, we recognize that refactorings are behavior-preserving due to good engineering and not because of any mathematical guarantee. It is the responsibility of the refactoring designer to identify all enabling conditions necessary to ensure that behavior is preserved. This important issue is explored further in Tokuda (1999).

5.3. Future research

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

Validation. How do we know a refactored program is correct? How do we know any program is correct? This is a hard problem in any circumstance. Lacking proofs of correctness and proofs of behavior preservation, we expect refactoring tools to be like compilers today: over time, we will learn to trust that the tool makes the correct transformations—until we have clearly reproducable bugs that tell us otherwise. As languages become simpler (see Section 5.4), there is hope that more progress can be made.

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 transformation 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.

Transactional refactorings. A related issue is the notion of what we will call transactional refactorings. A *transactional refactoring* can be decomposed into a composition of primitive transformations that themselves may not be behavior preserving, but the net effect of the transactional refactoring *is* behavior preserving. The **delegate_method_across_object_boundary** refactoring is an example. Conceptually, it is equivalent to a **move_method_across_object_boundary** (which removes the method entirely from its original class), followed by a **create_method_acessor** (which reintroduces the method to the original class and delegates its execution to the moved method). Individually, these refactorings cannot be applied in sequence. If clients of the original class reference the target method, the enabling conditions of the **move_method** refactoring will prevent the method from being moved. Only if we consider the net result of both refactorings do we see that the resultant refactoring is correct.

We dealt with such situations by creating **new** refactorings, such as **delegate_method.** However, a better approach will be to define primitive refactorings and to define a series of these refactorings as an atomic rewrite.

Program families. 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 transformation system to check enabling conditions for all programs in 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. The situation is further complicated for C++ by conditional complication 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 are not dependent on the presence of other tools. In this form, they 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^{TM} 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++^{TM} would allow transformed code to be displayed immediately in open windows.

5.4. 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.[10] Tools are now being developed to aid in this process (Simonyi, 1995; Baxter and Pidgeon, 1997; Batory et al., 1998).

6. Related work

Griswold developed behavior-preserving transformations for structured programs written in Scheme (Griswold, 1991). The goal of this system was to assist in the restructuring of functionally decomposed software. Software architectures developed using the classic structured software design methodology (Yourdon and Constantine, 1979) 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 (Johnson and Foote, 1988). Lieberherr implemented these transformations in the Demeter object-oriented software environment (Lieberherr et al., 1991). 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 (1987), the design principles of Johnson and Foote (1988) and the design history of the UIUC Choices operating system (Maydany et al., 1989). An example application of refactorings is the creation of an abstract superclass (Opdyke, 1992). Refactorings are implemented for C++ (Tokuda and Batory, 1995; Scherlis, 1998; Schulz et al., 1998; Tokuda and Batory, 1999) and for Smalltalk (Roberts et al., 1997). Roberts offers Smalltalk-specific design criteria for a program transformation tool (Roberts et al., 1997). One criteria which also applies to C++ software is that users should be allowed to name new entities introduced through transformations.

Refactorings are shown to automate the addition of design patterns to object-oriented software systems (Tokuda and Batory, 1995; Roberts et al., 1997; Scherlis, 1998; Schulz et al., 1998; Tokuda and Batory, 1999). Refactorings also support the addition of Pree's (Pree, 1994) hot-spot meta patterns (Tokuda and Batory, 1999).

7. Conclusion

There are regular patterns by which designs of object-oriented applications evolve: schema transformations, design pattern microarchitectures, and the hot-spot-driven-approach. Many evolutionary changes can be viewed as program transformations which are automatable with object-oriented refactorings. Refactorings are superior to hand-coding because they check enabling conditions to ensure that a change can be made safely, identify all lines of source code affected by a change, and perform all edits. Refactorings allow design evolution to occur at the level of a class diagram and leave the code-level details to automation.

Designs should evolve on an if-needed basis:

- "Complex systems that work evolved from simple systems that worked."—Booch
- "Start stupid and evolve."—Beck

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 limitations 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.

Acknowledgments

We thank the referees for their comments on earlier drafts of this paper. The final paper was much improved.

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

Notes

1. We use a limited definition of the term *design* referring to the aspect of design reflected in the extended class diagram notation from Gamma et al. (1995).
2. The seven refactorings which are not supported are: changing the value of a class variable, changing the code of a method, changing the default value of an instance variable, changing the inheritance parent of an instance

variable, changing the inheritance of a method, adding a method, and changing the order of superclasses. The first three refactorings are not behavior-preserving. The next two are not supported by mainstream object-oriented programming languages. The sixth (adding a method) cannot be automated. The seventh (changing the order of superclasses) is not supported because this research focusses on applications without multiple inheritance.

3. A Circle factory is created (Tokuda and Batory, 1995). Strategies are added (Section 3.2). The Bridge pattern is applied (Section 3.2.2). Finally, a decorator is added (Section 3.2).
4. We consider the 1 : N connection composition to be fundamental to a design. For this pattern, a template object is linked to a collection of hook objects. This implies that the template method has knowledge about how to use multiple hook methods and thus cannot be derived from the 1 : 1 connection composition in which the template method is coded for a single hook method.
5. 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.
6. The Command design pattern objectifies an action. The action is triggered by calling an `Execute()` method implemented in each derived class (Gamma et al., 1995).
7. This number is large because AUIS used 800 actions implemented as procedures and the conversion of a procedure to a command required a transformation. More refactorings did not imply more complexity. We found it was easier to choose the refactorings for AUIS than for CIM Works because conceptually, the evolution of AUIS's design required only five steps.
8. Many design patterns use runtime composition versus inheritance as an extension mechanism (Gamma et al., 1995). The dynamic nature of composition precludes static typechecking.
9. **Inherit** destroys this property since aggregates cannot have supervlasses.
10. When we began our work, tool support and availability of large Java files were nonexistent. This is no longer true today.

References

Banerjee, J. and Kim, W. 1987. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the ACM SIGMOD Conference*.
Batory, D. et al. 1998. JTS: Tools for implementing domainspecific languages. In *5th International Conference on Software Reuse*, Victoria, Canada. June 1998.
Baxter, I. and Pidgeon, C. 1997. Software change through design maintenance. In *Proceedings of the International Conference on Software Maintenance '97*, IEEE Press.
Bergstein, P. 1991. Object-Preserving class transformations. In *Proceedings of OOPSLA '91*.
Berstein, P. 1991. Object-preserving class transformations. In *Proceedings of OOPSLA '91*.
Budinsky, F.J. et al. 1996. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2).
Ellis, M. and Stroustrup, B. 1990. *The Annotated C++ Reference Manual*. Reading, Massachusetts: Addison-Wesley.

Florijn, G., Meijers, M., and van Winsen, P. 1997. Tool support for object-oriented patterns. In *Proceedings, ECOOP '97*, Sprinter-Verlag, Berlin, pp. 472–495.

Gamma, E. et al. 1993. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings, ECOOP '93*, Springer-Verlag, Berlin, pp. 406–421.

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

Gamma, E. et al. 1996. TUTORIAL 29: Design patterns applied. In *OOPSLA '96 Tutorial*.

Goldberg, A. 1984. *Smalltalk-80: The Interactive Programming Environment*. Reading, Massachusetts, Addison-Wesley.

Griswold, W. 1991. Program Restructuring as an Aid to Software Maintenance. Ph.D. thesis, University of Washington.

Johnson, R. and Foote, B. 1988. Designing reusable classes, *Journal of Object-Oriented Programming*, pp. 22–35.

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

Kim, J. and Benner, K. 1996. An experience using design patterns: lessons learned and tool support. *Theory and Practice of Object System*, 2(1):61–74.

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

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

Linton, M. 1992. Encapsulating a C++ library. In *Proceedings of the 1992 USENIX C++ Conference*, Portland, Oregon, pp. 57–66.

Maydany, P. et al. 1989. A class hierarchy for building stream-oriented file systems. In *Proceedings of ECOOP '89*, Nottingham, UK.

McGuire, P. 1997. Lessons learned in the C++ reference development of the SEMATECH computer-integrated manufacturing (CIM) applications framework. In *SPIE Proceedings*, 2913: pp. 326–344.

Morris, J.H. et al. 1986. Andrew: A distributed personal computing environment. *Communications of the ACM*.

Opdyke, W.F. 1992. Refactoring object-oriented-frameworks. Ph.D. thesis, University of Illinois.

Opdyke, W.F. and Johnson, R.E. 1993. Creating abstract susperclasses by refactoring. In *ACM 1993 Computer Science Conference*, Feb. 1993.

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

Pree, W. 1994. Meta Patterns—A Means for capturing the essentials of reusable object-oriented design. In *Proceedings, ECOOP '94*, Springer-Verlag, Berlin.

Pree, W. and Sikora, H. 1995. *Application of Design Patterns in Commercial Domains*. OOPSLA '95 Tutorial 11, Austin, Texas.

Pressman, R. 1992. *Software Engineering A Practitioner's Approach*. New York: McGraw Hill.

Roberts, D., Brant, J., Johnson, R. 1997. A refactoring tool for smalltalk. In *Theory and Practice of Object Systems*, 3(4).

Scherlis, W. 1998. Systematic change of data representation: program manipulations and case study. In *Proceedings of ESOP, '98*.

Schulz, B. et al. 1998. On the computer aided introduction of design patterns into object-oriented systems. In *Proceedings of the 27th TOOLS Conference*, IEEE CS Press.

Simonyi, C. 1995. The death of computer languages, the birth of intentional programming. *NATO Science Commitee Conference*.

Stewart, S. 1995. *Roadmap for the Computer-Integrated Manufacturing Application Framework*. NISTIR 5697.

Tokuda, L. and Batory, D. 1995. Automated software evolution via design pattern transformations. In *Proceedings of the 3rd International Symposium on Applied Corporate Computing*, Monterrey, Mexico, Oct. 1995.

Tokuda, L. and Batory, D. 1999. Automating three modes of object-oriented software evolution. In *Proceedings, COOTS '99*.

Tokuda, L. and Batory, D. 1999. Evolving object-oriented designs with refactorings. Technical Report TR99-09, Department of Computer Science, University of Texas.

Tokuda, L. 1999. Design evolution with refactorings. Ph.D. thesis, University of Texas.

Weinand A., Gamma E., and Marty R. 1988. ET++—An Object-Oriented application framework in C++. In *Object-Oriented Programming Systems, Languages, and Applications Conference*, San Diego, California, pp. 46–57, Sept. 1988.

Winsen, P.V. 1996. (Re)engineering with object-oriented design patterns. Master's Thesis, Utrecht University, INF-SCR-96–43.

Yourdon, E. and Constantine, L. 1979. *Structured Design*. Englewood cliffs, New Jersey: Prentice Hall.