5th Conference on Object-Oriented Technologies (COOTS), May 1999

Automating Three Modes of Evolution for Object-Oriented Software Architectures

Lance Tokuda, Don Batory

Department of Computer Science
University of Texas at Austin
Austin, TX 78712-1188
{unicron, dsb}@cs.utexas.edu

Abstract[1]

Architectural evolution is a costly yet unavoidable consequence of a successful application. One method for reducing cost is to automate aspects of the evolutionary cycle when possible. Three kinds of architectural evolution in object-oriented systems are: schema transformations, the introduction of design pattern microarchitectures, and the hot-spot-driven-approach. This paper shows that all three can be viewed as transformations applied to an evolving design. Further, the transformations are automatable with refactorings — behavior-preserving program transformations. A comprehensive list of refactorings used to evolve large applications is provided and an analysis of supported schema transformations, design patterns, and hot-spot meta patterns is presented. Refactorings enable the evolution of architectures on an if-needed basis reducing unnecessary complexity and inefficiency.

1 Introduction

All successful software applications evolve [Par79]. During the 1970s, evolution and maintenance accounted for 35 to 40 percent of the software budget for an information systems organization. This number jumped to 60 percent in the 1980s. It was predicted that without a major change in approach, many companies will spend close to 80 percent of their software budget on maintenance [Pre92]. As applications evolve, so do their architectures. Architectures evolve for multiple reasons:

- **Capability** — to support new features or changes to existing features.
- **Reusability** — to carve out software artifacts for reuse in other applications.
- **Extensibility** — to provide for the addition of future extensions.
- **Maintainability —** to reduce the cost of software maintenance through restructuring.

We have observed that architectures also evolve for human reasons:

- **Experience.** Experienced employees can often design a better architecture based on their knowledge of the current architecture.
- **New Perspective.** New project members often have new ideas about how an architecture could or should be structured. Many organizations use a code ownership model which empowers new employees with the ability to realize their new designs.

While motivations vary, the methods used for evolving architectures appear to follow regular patterns, particularly for object-oriented applications. Three kinds of object-oriented architectural evolution are: schema transformations, the introduction of design pattern microarchitectures, and the hot-spot-driven-approach. *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]. The *hot-spot-driven-approach* is based on the identification of aspects of a software program which are likely to change from application to application (i.e. *hot-spots*) [Pre95]. Architectures using abstract classes and template methods are prescribed to keep these hot-

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.

spots flexible.

Refactorings are behavior-preserving program transformations which directly aid in the implementation of new architectures. Primitive refactorings perform simple edits such as adding new classes, creating instance variables, and moving instance variables up the class hierarchy. Compositions of refactorings can create abstract classes, capture aggregation and components [Opd92], extract template and hook methods, and even install design pattern microarchitectures [Tok95]. Although composing refactorings to achieve a desired result may require some planning, this effort is negligible compared to the manual task of identifying all lines of source code affected by a change, performing hand-edits, retesting all fixes, and risking the introduction of new errors.

We are pursuing two approaches to promote refactoring research. The first is to evaluate refactorings of large applications. We believe that we are the first to provide empirical evidence on the usefulness of refactorings when applied to non-trivial applications [Tok99]. In one example, a major architectural change — the splitting of a class hierarchy — is automated[2]. In a second example on a large application (~500K LOCS), approximately fourteen thousand lines of code changes between two code releases are automated with refactorings. (See paper for details.)

A second approach to promoting refactoring research is to demonstrate that common forms of architectural evolution can be automated. This paper catalogs the schema transformations, design pattern restructurings, and hot-spot meta patterns which can be automated with refactorings. By demonstrating broad coverage of common modes of evolution, it is argued that refactorings will be generally useful in the evolutionary maintenance cycle.

A summary of the class diagram notation used throughout the remainder of this paper is presented in Figure 1. Within the main body of text, we use the following conventions:

- **Refactoring** — a refactoring.
- ***AbstractClass*** — an abstract class name.

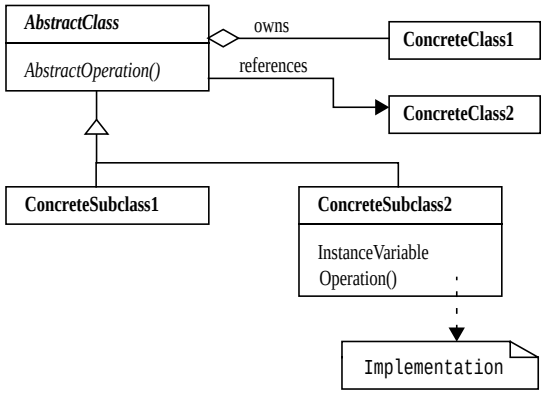

Figure 1: Notation

- **ConcreteClass** — a concrete class name.
- `Method()` — a method or procedure name.
- `Instance_variable` — an instance variable.

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 2. Applying refactorings is superior to hand-coding similar changes because it allow a designer to evolve the architecture of an existing body of code at the level of a class diagram leaving the code-level details to automation.

Banerjee and Kim proposed a set of schema evolutions for evolving object-oriented database schemas [Ban87] and Opdyke proposed a list of primitive refactorings for object-oriented languages [Opd92]. Roberts implements many of these refactorings for the Smalltalk language [Rob97]. In addition to previous refactorings, we have

2. The term *automated* in this paper refers to a refactoring's programmed check for enabling conditions and its execution of all source code changes. The choice of which refactorings to apply is always made by a human.

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**. `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.

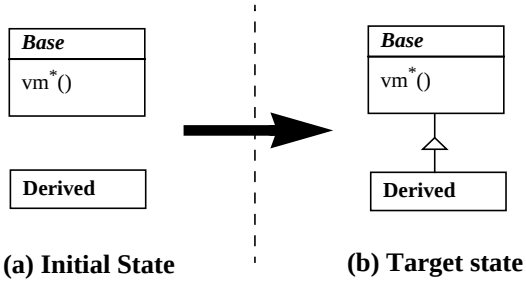

Figure 2: Inherit[Base, Derived] transformation

found that transforming actual applications requires a larger set. We enlarged the set of schema evolutions to include, for example, **substitute**. **Substitute** changes a class' dependency on some class **C** to a dependency on a superclass of **C** [Tok95]. A second new set of refactorings is language-specific. **Procedure_to_method** and **structure_to_class** are used to convert C artifacts to their C++ equivalents. A third set supports the addition of design pattern microarchitectures in evolving programs [Tok95]. An example is **add_factory_method** which creates a method returning a new object and replaces all C++ invocations of "`new Object`" with a call to the method. This refactoring is used to add the Factory Method design pattern [Gam95].

A list of refactorings used in our research is presented in Table 1. Refactorings proposed in previous work are

Schema Refactorings	
add_variable	move_method_across_ object_boundary
create_variable_accessor	extract_code_as_method
create_method_accessor	*declare_abstract_method*
rename_variable	*structure_to_pointer*
remove_variable	
push_down_variable	C++ Refactorings
pull_up_variable	*procedure_to_method*
move_variable_across_ object_boundary	*structure_to_class*
create_class	Design Pattern Refactorings
rename_class	*add_factory_method*
remove_class	*create_iterator*
inherit	*composite*
uninherit	*decorator*
substitute	*procedure_to_command*
rename_method	*singleton*
remove_method	1. [Ban87]
push_down_method	2. [Opd92]
pull_up_method	3. [Tok95]
	4. [Rob97]

Table 1: Object-oriented refactorings

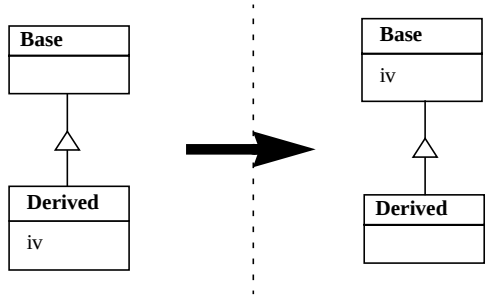

Figure 3.1: Using pull_up_variable to move instance variables "iv" from Derived to Base

noted. Refactorings first implemented by our research for object-oriented software evolution appear in italics.

3 Automatable Modes of Evolution

3.1 Schema Transformations

The database schema for an object-oriented database management system (OODBMS) looks like a class diagram for an object-oriented application. Similarly, OODBMS schema transformations have parallels in object-oriented software evolution. An example schema transformation is moving the domain of an instance variable up the inheritance hierarchy Figure 3.1. 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[3]:

Description from Banerjee and Kim [Ban87]	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.

Three other useful schema transformations not listed in [Ban87] are:

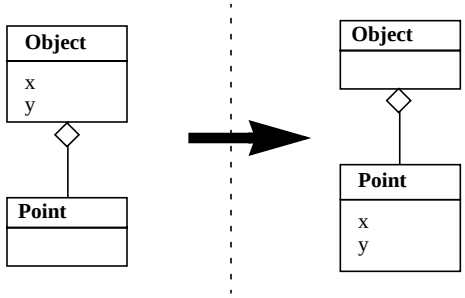

Figure 3.2: Using move_variable_across_object_b move instance variables x and y

Description	Refactoring
Move a variable through a composite link	**move_variable_across_object_boundary** (**Figure 3.2**)
Move a method through a composite link	**move_method_across_object_boundary**
Change a class' dependency on a class **C** to a dependency on a superclass **S** of **C**	**substitute (Figure 3.3)**

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 architectures.

3. 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 is currently limited to applications without multiple inheritance.

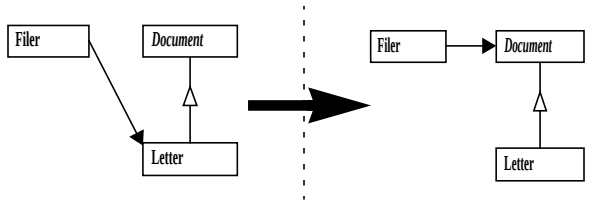

Figure 3.3: Using substitute to change Filer's reference to a Letter to a reference to a Document

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 [Gol84], ET++ [Wei88], MacApp [App89], and InterViews [Lin92]. 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.	[Tok99]
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.	[Tok95]
Singleton	Singleton ensures a class will have only one instance and provides a global point of access to it. The **singleton** refactoring converts an empty class into a singleton.	[Tok99]

We directly support three additional patterns as refactorings:

Pattern	Description
Composite	Composite composes objects into tree structures to represent part-whole hierarchies. 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.
Iterator	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 [Gam93]. 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 of related or dependent objects without specifying their concrete class.	[Tok95]

Pattern	Description	Example
Visitor	Visitor lets you define a new operation without changing the classes of the elements on which it operates.	[Rob97]

At least five additional patterns from [Gam95] 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 independently from the clients that use them.
Template Method	Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure.

In all cases, we can apply refactorings to simple designs to create the designs used as prototypical examples in [Gam95]. 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 [Gam95] (Figure 3.4), the **TextShape** class adapts **TextView**'s `GetExtent()` method to implement `BoundingBox()`. The adapter can be constructed from the original **TextView** class (Figure 3.5) in five steps:

1. Create the classes **TextShape** and ***Shape*** using **create_class**.
2. Make **TextShape** a subclass of ***Shape*** using **inherit** (Figure 3.6).
3. Add the `text` instance variable to **TextShape** using **add_variable** (Figure 3.7).

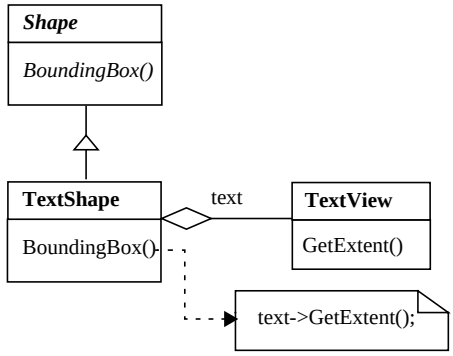

Figure 3.4: TextShape adapts TextView's interface

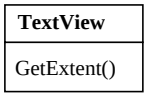

Figure 3.5: Unadapted TextView class

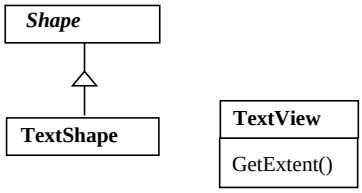

Figure 3.6: Adapter class created

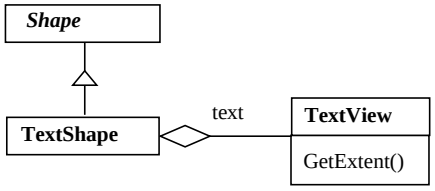

Figure 3.7: Adaptee instance variable added to adapter

4. Create the `BoundingBox()` method which calls `text->GetExtent()` using **create_method_accessor**. **Create_accessor_method** creates a method which replaces calls of the form `instance_variable->method()`.
5. Declare `BoundingBox()` in ***Shape*** using **declare_virtual_method** (Figure 3.4).

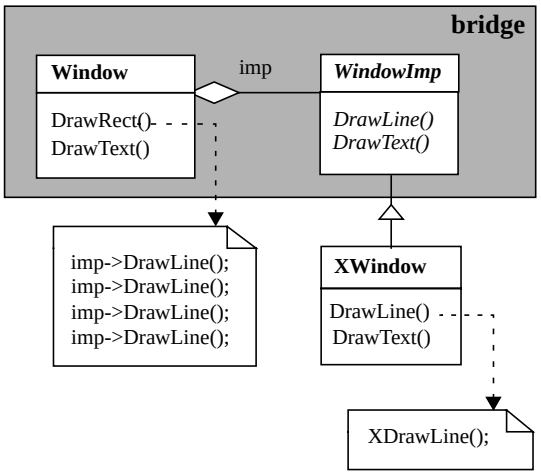

Figure 3.8: Bridge design pattern example

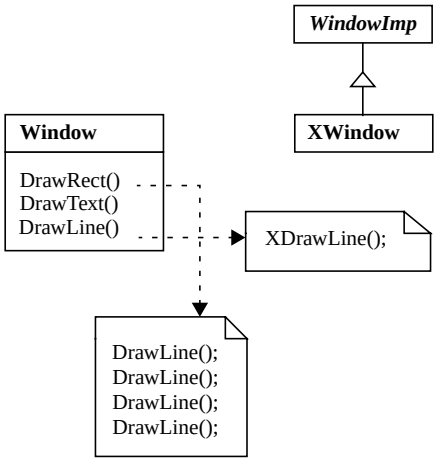

Figure 3.10: Implementor classes created

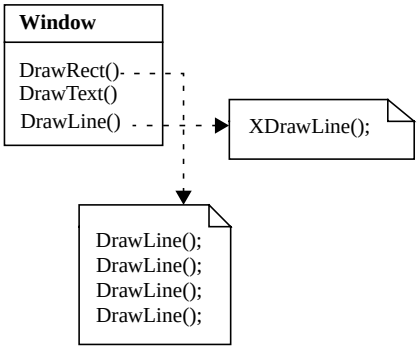

Figure 3.9: Design for a single window system

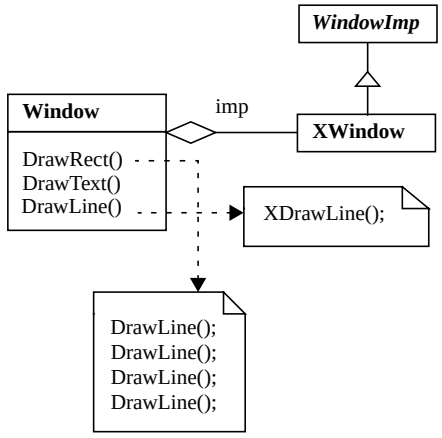

Figure 3.11: Implementor instance variable added to Window

3.2.2 Bridge

Bridge decouples an abstraction from its implementation so that the two can vary independently. In the example from [Gam95] (Figure 3.8), 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 3.9 depicts a system designed for X-Windows. This system can be evolved with refactorings to use the bridge design pattern in seven steps:

1. Create classes **XWindow** and ***WindowImp*** using **create_class**.
2. Make ***WindowImp*** a superclass of **XWindow** with **inherit** (Figure 3.10).
3. Add instance variable `imp` to the **Window** class using **add_variable** (Figure 3.11).
4. Move methods `DrawLine()` and `DrawText()` to the **XWindow** class using the refactoring **move_method_across_object_boundary**. These methods are accessed through the `imp` instance variable (Figure 3.12).
5. Declare method `DrawLine()` and `DrawText()` in ***WindowImp*** with **declare_abstract_method**.

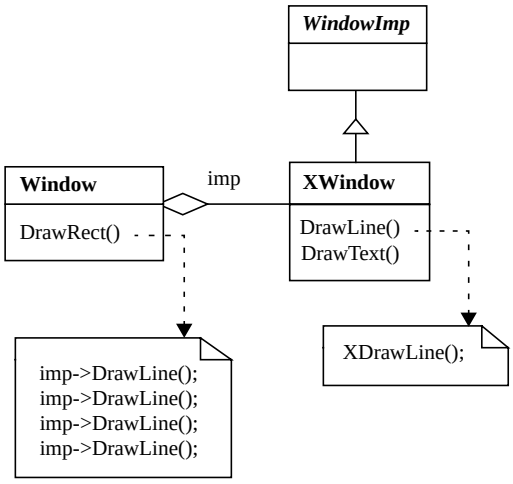

Figure 3.12: Window system specific methods moved to XWindow class

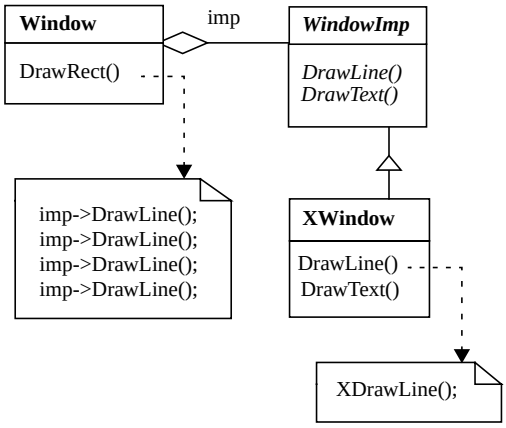

Figure 3.13: Virtual methods declared so that "imp" can be generalized to class *WindowImp*

6. Change the type of instance variable `imp` from **XWindow** to ***WindowImp*** using **substitute** (Figure 0.13).
7. Add a `DrawText()` method to **Window** which calls `DrawText()` in ***WindowImp*** using **create_method_accessor** (Figure 3.8).

The Bridge architecture 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 [Gam95, pp. 18-20]. Refactorings allow a designer to safely migrate from statically checkable designs using inheritance to dynamically defined designs using object-composition.

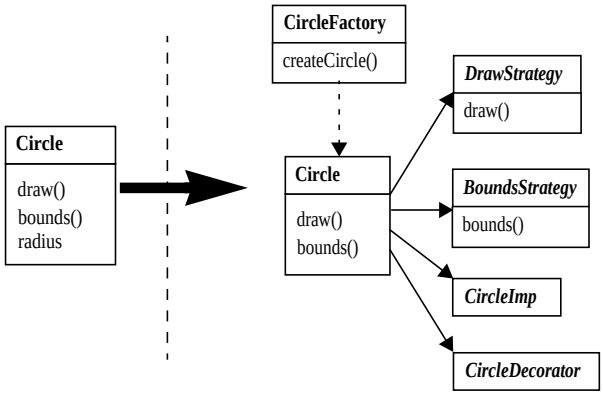

Figure 3.14: Overenthusiastic use of design patterns

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 [Gam96] is displayed in Figure 3.14. 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 that what is actually required. The migration from a single **Circle** class to the complex microarchitecture in Figure 3.14 can be viewed as a transformation. This transformation is in fact automatable with refactorings[4]. 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 architecture because there is no refactoring enabled evolutionary path which leads to their use. Refactorings allow a designer to focus on fundamental patterns when

4. A Circle factory is created [Tok95]. Strategies are added (Section 3.2). The Bridge pattern is applied (Section 3.2.2). Finally, a decorator is added (Section 3.2).

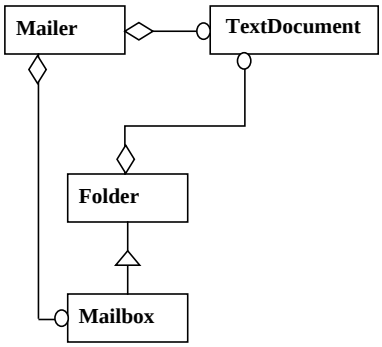

Figure 3.15: Initial state of mailing system

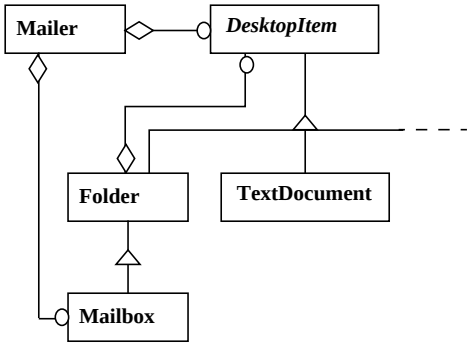

Figure 3.16: Final state of mailing system

creating a new software architecture. Patterns supported through refactorings can be added on an if-needed basis to the current or future architecture at minimal cost.

3.3 Hot-Spot Analysis

The *hot-spot-driven-approach* [Pre94] 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 the instance variables between applications are likely to differ, Pree prescribed the creation of abstract classes. Refactorings have repeatedly demonstrated the ability to create abstract classes [Opd93, Tok95, Rob97]. As an example, Pree and Sikora provide a Mailing System case study [Pre95]. Figure 3.15 displays the initial state of its software architecture. In this system, **Folder** cannot be nested, and only **TextDocument** can be mailed. Their suggested architecture is displayed in Figure 3.16. Under the improved architecture, **Folders** can be nested and any subclass of ***DesktopItem*** can be mailed. Refactorings can automate these changes in five steps:

1. Create a ***DesktopItem*** class using **create_class** (Figure 3.17).
2. Make ***DesktopItem*** a superclass of **TextDocument** using **inherit** (Figure 3.18).
3. Generalize the link between **Mailer** and **TextDocument** to a link between **Mailer** and ***DesktopItem*** using **substitute** (Figure 3.19). Subclasses of ***DesktopItem*** can now be mailed.

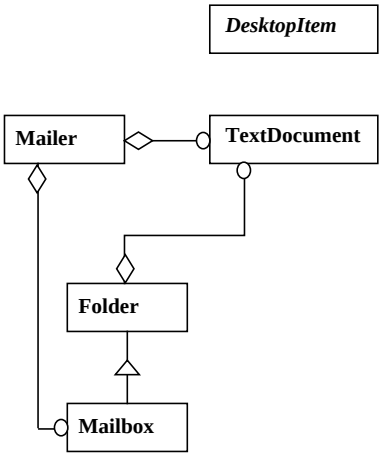

Figure 3.17: Empty TextDocument class created

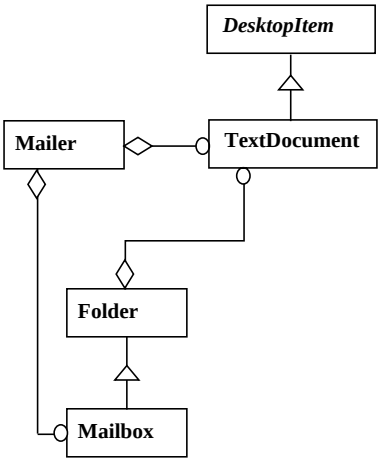

Figure 3.18: TextDocument inherits from *DesktopItem*

4. Generalize the link between **Folder** and

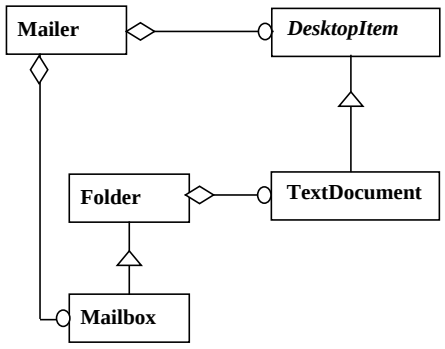

Figure 3.19: Mailer dependency changed from TextDocument to *DesktopItem*

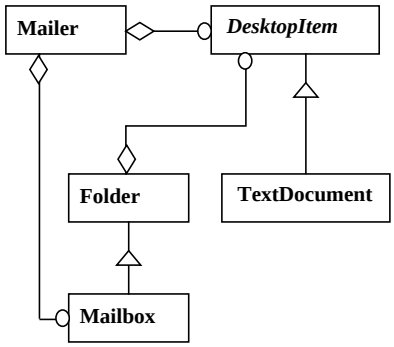

Figure 3.20: Folder can contain any *DesktopItem*

TextDocument to a link between **Folder** and ***DesktopItem*** using **substitute** (Figure 3.20). **Folder** can now contain any ***DesktopItem***.

5. Make **Folder** a subclass of ***DesktopItem*** using **inherit** (Figure 3.16). A **Folder** which can contain a ***DesktopItem*** can now contain another **Folder**.

With the improved architecture, a **Folder** can be nested within another **Folder** and ***DesktopItem*** provides a superclass for adding other types of media to be mailed. These changes which would normally be implemented and tested by hand can be automated with refactorings.

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 3.21 is an example of a template method and hook method defined in the same class. Different subclasses of ***T*** can override hook

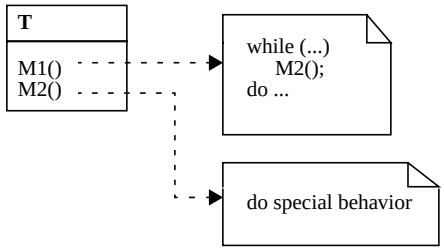

Figure 3.21: Template and hook methods in same class

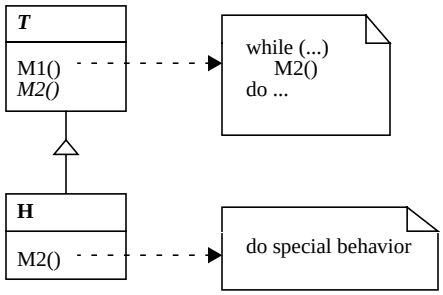

Figure 3.22: Hook method M2() overridden in class H

method M2() which leads to differing functionality in template method M1(). (Figure 3.22). Pree identifies seven meta patterns for template and hook methods: unification, 1:1 connection, 1:N connection, 1:1 recursive connection, 1:N recursive connection, 1:1 recursive unification, and 1:N recursive unification [Pre94]. Refactorings automate the introduction of meta patterns into evolving architectures. The transitions between patterns enabled by refactorings are displayed in Figure 3.23[5]. As examples, we demonstrate support for the first two transitions.

In the unification composition, both the template and hook methods are located in the same class (Figure 3.21). The behavior of the template is changed by overriding the hook method in a subclass (Figure 3.22). An architecture with no template or hook methods can be transformed to use the unification meta pattern (transition 1 from Figure 3.23). Consider the class

5. We consider the 1:N connection composition to be fundamental to an architecture. 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.

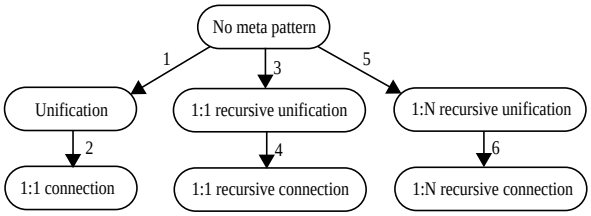

Figure 3.23: Hot-spot meta pattern transitions enabled by refactorings

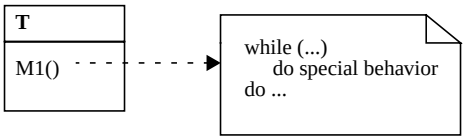

Figure 3.24: Method M1() calls a special behavior which differs for each application

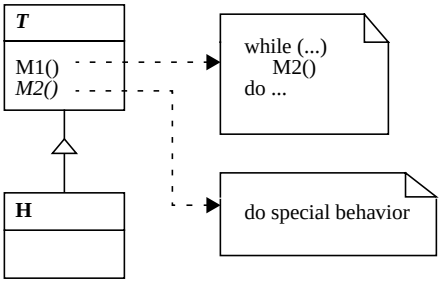

Figure 3.25: Hook class created

diagram in Figure 3.24 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 3.21). **Extract_code_as_method** replaces a block of code with a call to a newly created method which executes the block.

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

1. Create class **H** using **create_class**.

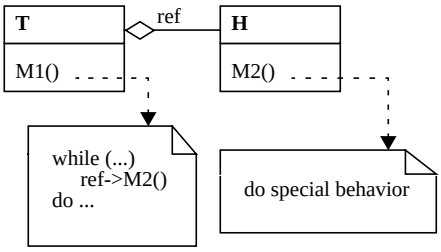

Figure 3.26: 1:1 connection

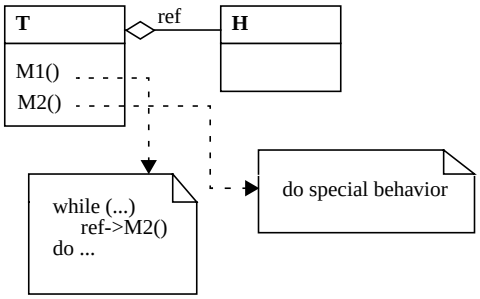

Figure 3.27: Connection to H object created

2. Make ***T*** a superclass of **H** using **inherit** (Figure 3.25).
3. Make *M2()* overridable by the subclasses of ***T*** using **declare_abstract_method**.
4. Move the implementation of M2() into **H** using **push_down_method** (Figure 3.22).

As a second example, we support the transition from unification to 1:1 connection (transition 2 from Figure 3.23). Consider the 1:1 connection meta pattern which stores the hook method in an object owned by the template class (Figure 3.26). 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 3.21) as a starting point.

1. Create class **H** using **create_class**.
2. Add an instance variable `ref` to **T** with **add_variable** (Figure 3.27).
3. Move M2() to class **H** using **move_method_across_object_boundary** (Figure 3.26).

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 3.23 are similarly supported.

3.3.3 Role of Refactorings for Hot-Spot Analysis

The hot-spot-driven-approach provide 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 architectures. 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 demonstrates 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 Related Work

Griswold developed behavior-preserving transformations for structured programs written in Scheme [Gri91]. 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 [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 architectures 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. Bergstein's transformations are object preserving so they cannot add, delete, or move methods or instance variables exported by a class.

Banerjee and Kim identified a set of schema transformations which accounted for many changes to evolving object-oriented database schema [Ban87]. Opdyke defined a parallel set of behavior-preserving transformations for object-oriented applications based on the work by Banerjee and Kim, the design principles of Johnson and Foote [Joh88], and the design history of the UIUC Choices software system [May89]. These transformations were termed *refactorings*. Roberts developed the Smalltalk Refactory Browser which implements many of these refactorings [Rob97].

Tokuda and Batory proposed additional refactorings to support design patterns as targets states for software restructuring efforts [Tok95]. Refactorings are shown to support the addition of design patterns to object-oriented applications [Tok95, Rob97, Sch98]. Winsen used refactorings to make design patterns more explicit [Win96]. Tokuda and Batory demonstrated that refactorings can automate significant (greater than 10K lines of code) changes when applied to real applications [Tok99].

A number of tools instantiate a design pattern and insert it into existing source code [Bud96, Kim96, Flo97]. Instantiations are not necessarily refactorings, so testing of any changes may be required. Florijn and Meijers check invariants governing a pattern and repairs violations when possible. Refactorings do not have this pattern-level knowledge.

5 Summary

Architectural evolution is a costly yet unavoidable consequence of a successful application. One method for reducing cost is to automate aspects of the evolutionary cycle when possible. For object-oriented applications in particular, there are regular patterns by which architectures evolve. Three modes of architectural evolution are: schema transformations, the introduction of 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 architectural evolution to occur at the level of a class diagram and leave the code-level details to automation.

Architectures should evolve on an if-needed basis:

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

Refactorings directly address the need to evolve from simple to complex designs by automating many common design transitions. We believe that the majority of all object-oriented applications undergoes some form of automatable evolution. The broad scope of supported changes indicates that refactorings can have a significant impact when applied to evolving designs. This claim is validated with real applications in [Tok99] where many hand-coded changes between two major releases of two software systems are automated.

The limiting factor barring the widespread acceptance of refactoring technology appears to be the availability of production quality refactorings for the two most popular object-oriented languages: C++ and Java. Our current research identifies implementation issues for C++ [Tok99].

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.

[Ber91] P. Bergstein. Object-Preserving Class Transformations. In *Proceedings of OOPSLA '91*, 1991.

[Bud96] F. J. Budinsky et.al., Automatic Code Generation from Design Patterns. In *IBM Systems Journal*, Volume 35, No. 2, 1996.

[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.

[Gam96] E. Gamma et. al. *TUTORIAL 29: Design Patterns Applied.* OOPSLA '96 Tutorial, 1996.

[Gol84] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, Massachusetts, 1984.

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

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

[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.

[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 *Proceedings, ECOOP '97*, pages 472-495, Springer-Verlag, 1997.

[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.

[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.

[Pre95] W. Pree and H. Sikora. *Application of Design Patterns in Commercial Domains*. OOPSLA '95 Tutorial 11, Austin, Texas, October 1995.

[Pre92] R. Pressman. *Software Engineering A Practitioner's Approach*, McGraw Hill, 1992.

[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.

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

[Tok99] L. Tokuda and D. Batory. *Evolving Object-Oriented Designs with Refactorings*. University of Texas, Department of Computer Science, Technical Report TR99-09, March 1999.

[Wei88] A. Weinand, E. Gamma, and R. Marty. ET++ -- An Object-Oriented Application Framework in C++. In Object-Oriented Programming Systems, Languages, and Applications Conference, pages 46-57, San Diego, California, September 1988.

[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.