Evolving Object-Oriented Designs with Refactorings

by

Lance Aiji Tokuda, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 1999

Evolving Object-Oriented Design with Refactorings

Approved by
Dissertation Committee:

_____

_____

_____

_____

_____

Acknowledgments

First and most importantly, I am deeply indebted to my advisor, Don Batory, for many years of support and advice, for providing constant direction and focus to my research, and for reading countless drafts of my dissertation and conference papers (paper16.fm is the record). Thanks Don for everything!

I would like to thank my family who has been holding off on my graduation celebration trip to Las Vegas for some years now. You can finally address me as 'doctor'.

I would like to thank the hundreds of employees at Resumix Inc. who worked towards its eventual sale to Ceridian Corporation. The life of a rich graduate student is more fun than most people can possibly imagine.

I would like to thank my best friend, Tina Sayama, for proofreading my dissertation, for those weekly five hour phone calls in my early years at UT, and for all the fun times we spent together. More than anything else, I wanted to finish my degree so I could return to California and be with you.

Finally, I would like to thank all of the volleyball players from my intramural championship teams: Grace (my favorite!), Lorinda, Faye, Heather, Julie, and dozens of others. Volleyball was the best part of my life at UT and I could never have played it alone. I also want to thank my intramural champion miniature golf partners: Jeff, Grace, and most of all Marina for that time we won by thirty strokes. Many people receive Ph.D.'s from the University of Texas but no one I know of has ever won seventeen intramural sports championships — this is what I am most proud of in the time I spent here at Texas. Thanks guys for making graduate school fun!

LANCE AIJI TOKUDA

The University of Texas at Austin

September 1999

Contents

Chapter 1

Introduction

1.1 Problem: design evolution

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 evolution and maintenance [Pre92].

As applications evolve, so do their designs. Designs evolve for many 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 designs also evolve for human reasons:

- **Experience.** Experienced employees may create better designs based on their domain knowledge.

- **New Perspective.** New project members often have different ideas about how a design could or should be structured. Many organizations use a code ownership model which empowers new employees with the ability to realize their ideas.
- **Experimentation.** Arriving at a suitable design may require exploration of different design paths. We have observed software cycles in which the principal development activity was experimentation with multiple designs.

It is well-known that object-oriented design methodologies offer important opportunities for reducing maintenance costs: the modularity of classes and frameworks can simplify reuse and extension. Language features such as inheritance also contribute to maintenance by allowing specializations of a class to be built without altering the original class.

Evolving a paper design for an unimplemented software application is relatively easy[1]. Tools exist for constructing and editing an application's class diagram. Inheritance and aggregation relationships can be created and deleted, instance variables can be moved up and down the inheritance hierarchy, and classes can be added and deleted, all through a point and click interface.

Evolving the design of a legacy system is much more difficult. Besides editing the paper design, an engineering team must also alter an application's source code to reflect the new design. Conditions must be checked to ensure that design changes can be made safely, lines of affected source code are identified, changes are coded, the system is tested to check for the introduction of errors, any errors are fixed and the system is retested. Retesting continues until the expected likelihood of an error occurring is sufficiently low.

This process can be especially difficult to execute during a software cycle in

1. Here we are referring to the task of changing design diagrams and documents to reflect the new design, not the intellectual task of determining what the new design should be.

which new development requires a stable design. One option is to branch the code hierarchy to pursue both a new design and parallel development under the stable older design. This software process requires a tedious and possibly costly integration of all changes under both designs. An option which avoids the integration phase is to suspend affected development while a new design is being implemented. Under this option, design changes may become a project bottleneck. Experimentation with different designs can be very costly, and delays in the successful completion of design tasks may lead to delays in the entire project.

Our research seeks to reduce the effort required to evolve and experiment with designs by automating the process when feasible. Ideally, designers would edit a graphical design and any implied changes would be immediately reflected in source code through automation.

1.2 Characteristics of an acceptable solution

A major focus of this thesis is to develop technology which can be transferred to a mainstream programming environment. An acceptable solution should have the following characteristics:

- **Support for a mainstream programming language.** We recognize three mainstream programming languages — C, Cobol, Fortran, and their derivatives. For object-oriented software development, C++ (a derivative of C) is the overwhelming choice in industry. The Java language whose syntax is similar to C++ is also generates significant interest because of its applicability to the internet.

- **Support for legacy applications.** The assets of a software organization reside largely in its legacy code base. The primary function of most software organizations is to maintain and enhance this legacy. A solution would be most useful if it could assist in this function.
- **Support for large applications.** A solution should scale to meet the needs of today's larger applications. Applications containing over 100K lines of source code are commonplace.

1.3 Approach: refactoring

Refactorings are behavior-preserving program transformations which can aid in the creation of new designs[2] and the restructuring of legacy designs [Opd92]. Primitive refactorings perform simple edits on a class diagram such as adding new classes, creating instance variables, and moving instance variables up and down the class hierarchy. Compositions of refactorings have been shown to create abstract classes and capture aggregation and components.

One of the refactoring complexities recognized by Opdyke in 1992 was that there was no systematically organized explanation for the kinds of design changes that people make [Opd92]. Without any kind of organization, it was difficult to determine if the capabilities offered by refactorings would be generally useful in the evolution of real-world designs.

This changed with the birth of the patterns movement which sought to capture solutions to common object-oriented design problems [Gam95, Pre95]. Patterns were used successfully for the purpose of restructuring existing designs [Hun95]. Furthermore, we showed that many patterns can be viewed as target states of

2. The term *design* as used in this paper is a limited definition which refers to an application's class diagram with extensions from Gamma [Gam95]. The term is closely aligned with database schema design. For programs, state diagrams, object interaction diagrams, and scenarios all capture aspects of an application's design which are not visible on a class diagram.

automated[3] refactorings applied to evolving designs [Tok95, Tok99].

This research assesses the capabilities of refactorings for evolving object-oriented designs and attempts to determine if refactoring technology can be successfully transferred to the mainstream. To that end, we pursue the following goals:

1. to identify a set of refactoring capabilities which can be used to evolve object-oriented designs,
2. to design and code a set of refactorings which implement the capabilities for a mainstream programming language,
3. to test the refactorings on a small scale,
4. if successful, to test on a large scale, and
5. to identify the benefits, limitations, requirements, and open issues when transferring refactoring technology to a mainstream programming environment.

The next section describes the contents of this thesis as they relate to these goals.

1.4 Overview

Chapter 2 identifies related work in refactoring, program transformations, tools, and patterns.

Chapter 3 introduces refactorings. This chapter defines a refactoring, provides an example, identifies refactorings contributed by this research, and discusses the issue of behavior preservation.

Chapter 4 catalogs the schema transformations, design patterns, and hot-spot

3. 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.

meta patterns whose introduction can be automated with refactorings. Refactoring support for design patterns and hot-spot meta patterns is an original contribution of this research.

Chapter 5 provides a proof-of-concept example which demonstrates the ability of refactorings to add design patterns to an evolving application. The results motivate further research on real-world applications.

Chapter 6 details the refactoring of two real-world applications. This chapter provides application selection criteria, methodology for applying refactorings, and a step-by-step description of refactorings applied. The applications are (to our knowledge) the largest transformed to date.

Chapter 7 describes the refactoring design and implementation. It also includes lessons learned for future refactoring implementors.

Chapter 8 is an evaluation of this work. Sections included discuss benefits, limitations, requirements, and open issues.

Chapter 9 presents a summary of our research and results.

1.5 Notation

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

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

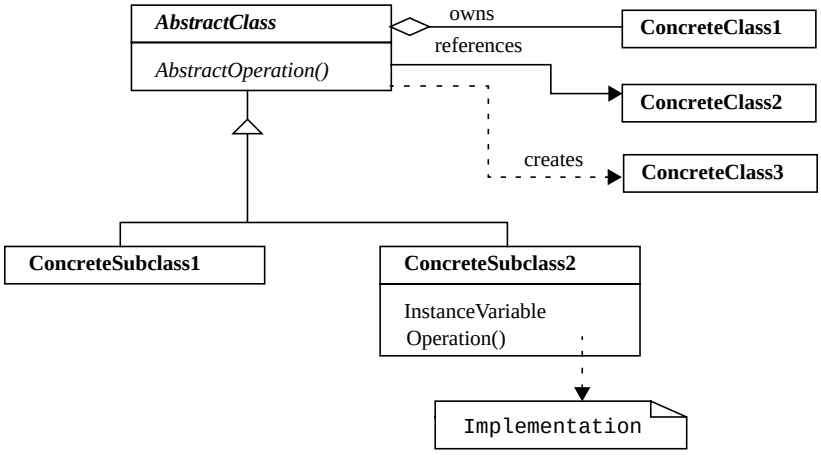


Figure 1: Notation

Chapter 2

Related Work

Our research applies transformation systems to recent developments in object oriented design and evolution. This chapter highlights the significant contributions impacting this research.

2.1 Transformation systems

This work distinguishes between specification-to-source and source-to-source transformations. Specification-to-source transformations transform a high level program specification to compilable source. Examples are software generators [Rea86, Bax90, Bat94]. The ratio between lines of specification to lines of code generated can be one-to-ten or higher. The popularity of software generators is limited because they are often domain-specific and they require the use of non-standard programming languages.

Source-to-source transformations transform a program coded in a given language to a new program coded in the same language. They are not limited to a domain and the transformations can be written to support standard programming languages. The benefits of source-to-source transformations have yet to be quantified. Refactorings are a form of behavior-preserving source-to-source transformation.

2.2 Transforming structured programs

Griswold developed behavior-preserving transformations for programs written in Scheme [Gri91]. An example transformation is **var-to-expr** which replaces occurrences of a variable with the expression it is bound to. The goal of these transformations 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. Structured designs represent programs as *structure charts* — trees whose nodes represent functionality and whose branches represent the transfer of control and data (Figure 8.1). The presence of control information makes it difficult to relocate subtrees of the structure chart since both input and output control information can be unique to a specific location in the chart. As a result, most of Griswold's transformations are limited to the level of a function or a block of code.

Baxter's DMS is a source-to-source transformation system used commercially on COBOL programs up to one million lines of code in size [Bax97]. The primary functions of Baxter's transformations are to unify duplicate code and remove dead code. The transformations do not preserve behavior.

2.3 Object-oriented designs

Before 1992, there were few notions about what a good object-oriented design should be. The most popular book on object-oriented design was the first edition of Booch's *Object-Oriented Analysis and Design with Applications* [Boo94]. The major contribution of this text was a method for documenting an application's

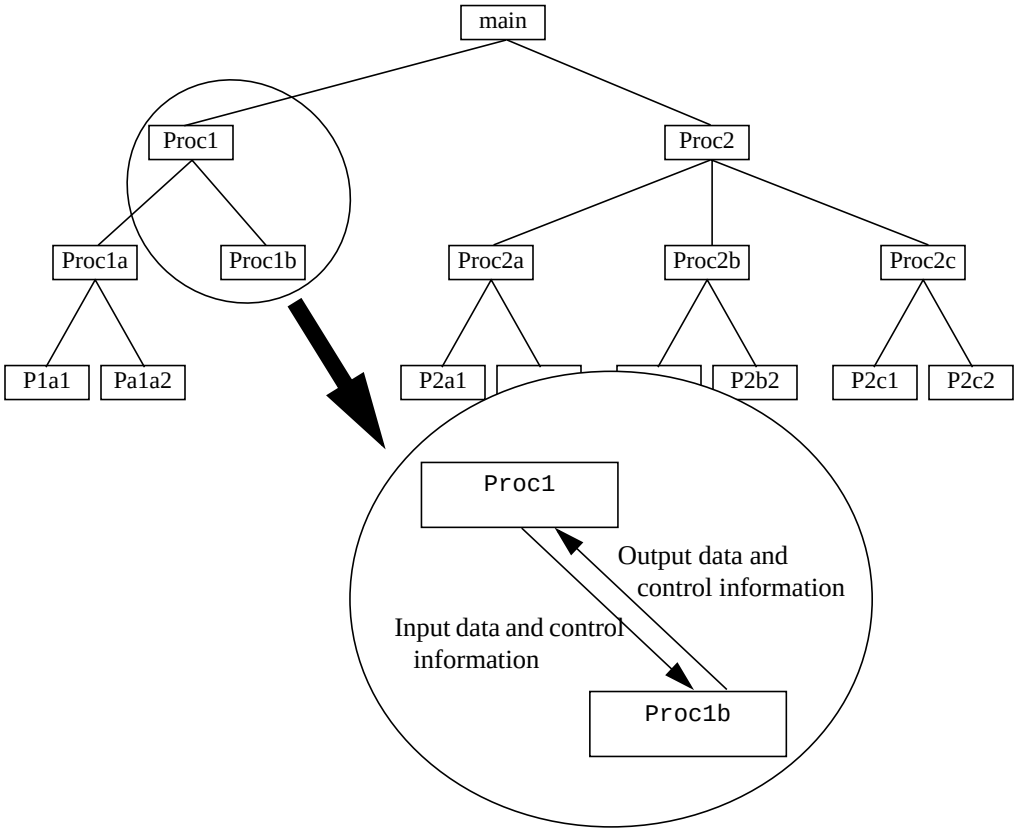


Figure 8.1: Structure chart for a traditional structured design. Control information for any node of the chart can be unique to that node making relocation of nodes difficult.

design. There was little guidance for creation and evolution of good designs. The primary method for transferring the knowledge of experienced object-oriented designers was through the publication of design rules of thumb:

- Rochat proposed guidelines for developing good programming style in Smalltalk [Roc86]. An example guideline is to carefully name classes and class members.
- Johnson and Foote proposed a set of rules for designing reusable classes [Joh88]. Example rules are to reduce the number of arguments of a method to fewer than six and to keep the size of methods small (fewer than thirty lines).

- Lieberherr and others proposed the “Law of Demeter” which states that a method should have limited knowledge of an object model [Lie89]. The effect is to prevent a method from being executed through more than one link (e.g. “`this.a.b.c.d.foo()`”). They note, however, that the law can require an increase in the number of methods and method arguments resulting in poor code readability and slower execution speeds [Lie88].

In general, rules provided a checklist of things to look for and things to avoid but offered limited assistance in solving any particular object-oriented design problem.

A new approach to the transfer of object-oriented design experience was presented in 1992 when Gamma released a preliminary version of a technical report which proposed design patterns as a method of capturing expert solutions to many common object-oriented design problems. Patterns were discovered in a wide variety of applications and toolkits including Smalltalk Collections [Gol84], ET++ [Wei88], MacApp [App89], and InterViews [Lin92]. This report sparked a patterns revolution in the object-oriented community and a flood of articles and books on patterns were subsequently published [Gam95, And94, Bec94, Coa95, etc.]. An interesting extension was the work of Pree who identified meta patterns which abstractly described how many design patterns worked [Pre94].

While design patterns were useful in the initial software design, they were often applied in the maintenance phase of the software lifecycle [Gam93]. Huni demonstrated the advantages of adding patterns to an existing design by evolving a framework for network protocols [Hun95]. This motivated work on tool support for patterns.

2.4 Transforming object-oriented designs

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*.

Implementations of Opdyke's refactorings for C++ were developed by Tokuda and Batory [Tok95, Tok99] and Schultz [Sch98b]. An implementation for Smalltalk was developed by Roberts [Rob97]. Roberts offered Smalltalk-specific design criteria for a program transformation tool. One criteria which also applied to C++ software is that users should be allowed to name new entities introduced through transformations.

Opdyke first claimed that a series of refactorings could be used to create abstract classes and part-whole relationships [Opd92, Opd93]. This was demonstrated for abstract classes [Tok95] and for part-whole relationships in the CIM Works example from Chapter 6. Scherlis proposed refactorings which were shown to support a hypothetical derivation of the Java `String` and `StringBuffer` classes from an original null terminated string class [Sch98a]. Tokuda and Batory proposed and implemented refactorings to support Gamma's design patterns [Tok95] and Pree's hot-spot meta patterns [Tok99] as target states for software

restructuring efforts.

Refactorings have been used alter existing designs. Winsen used refactorings (primarily renaming) to make design patterns more explicit [Win96]. Tokuda and Batory [Tok95], Roberts [Rob97], and Shultz [Sch98b] added design patterns to evolve an application's design.

Fowler proposed a set of manual refactorings for evolving an application's design. An example is **introduce_null_object** which replaces checks for a null object pointer with an object which performs the null case behaviors [Fow99]. Fowler's refactorings do not state the enabling conditions required to preserve behavior. Compilation and testing is recommended at multiple points within each refactoring.

2.5 Preserving behavior

Banerjee and Kim identified a set of invariants which preserve behavior for object-oriented database schema [Ban87] and Opdyke proposed a similar set of invariants to preserve behavior for refactorings [Opd92]. Opdyke's refactorings are accompanied by proofs which demonstrate that the enabling conditions he identified for each refactoring preserve the invariants.

Roberts used dynamic analysis to perform some enabling condition checks. An example is for the Smalltalk **rename_method** refactoring. Smalltalk allows dynamically created messages to be sent via the "perform: message". For an application using this approach, any automatic renaming process has the potential of failure. Robert's implementation of **rename_method** renamed the original method and replaced it with a method wrapper. As the program ran, the wrapper detected sites that called the original method. Whenever a call to the old method was detected, the method wrapper suspended execution of the program, traced up the call stack to the sender, and changed the source code to refer to the new,

renamed method [Rob97]. Roberts notes that the major drawback to this type of refactoring is that the refactoring is only as good as the test suite. If there are pieces of code that are not executed, they will never be analyzed, and the refactoring will not be completed for that particular section of code.

Hursch and Seiter presented a subset of Opdyke's refactorings which operated under an alternative framework for preserving behavior. The framework consisted of three parts: a formal description of the dependencies between the main components of an object-oriented framework, a definition of behavioral equivalence, and a process model for maintaining consistency and behavior between components [Hur96]. A refactoring implementation using the Hursch and Seiter framework is currently under development for an object-oriented version of Scheme.[1]

Chan and others propose promises to address the important topic of preserving behavior when full source for all components is not available [Cha98]. *Promises* are annotations which act as surrogates for actual components. They allow refactoring enabling conditions to be checked without viewing the original source code. Promises offer a possible solution to the problem of transforming proprietary third party components, however, the solution requires developers to define and maintain non-standard information about their components. The barriers to adopting this approach will be similar to those encountered when proposing major changes to a language standard, and thus may be difficult to overcome.

1. There is no provision in this formal framework to represent the syntax and semantics of the language being transformed. Programs are transformed at the level of a class diagram independent of the underlying language. We believe that this framework is subject to the same errors and oversights encountered with Opdyke's invariants since these problems only occur when certain language features are present (e.g. aggregates for C++).

2.6 Other tools

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. Also, the number of lines of code added by instantiating a pattern is generally small. Budinsky offers an array of implementation options for each pattern which would also be beneficial for refactorings. Florijn and Meijers check invariants governing a pattern and repairs violations when possible. Refactorings do not have this pattern-level knowledge.

A related approach for adding patterns is to provide language constructs which directly support a pattern's implementation. Bosch proposes language support for eight of Gamma's design patterns [Bos98]. Language constructs reduce pattern implementation costs and make patterns easier to recognize. The major drawback to this approach is that no standard language having the language features proposed currently exists.

Chapter 3

Refactorings

A *refactoring* is a parameterized behavior-preserving program transformation that typically has a straightforward (but not necessarily trivial) impact on an application's design. 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.

Refactorings check enabling conditions to ensure that program behavior is preserved, identify source code affected by a change, and execute all changes. Programs are restructured by applying a series of refactorings. When individual refactorings preserve behavior, a series of refactorings will also preserve behavior.

3.1 Example

An example refactoring is **inherit[*Base*, Derived]**, which establishes a superclass-subclass relationship between two classes, ***Base*** and **Derived**, that were previously unrelated. From the perspective of an object-oriented class diagram, the **inherit** refactoring merely adds an inheritance relationship between the ***Base*** and **Derived** classes, but also it alters the application's source code to reflect this change. The complete definition for **inherit[*Base*, Derived]** is given in Figure 2.1.

Name:

Inherit[*Base*, Derived]

Purpose:

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

Arguments:

Base - superclass name

Derived - subclass name

Description:

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

Enabling Conditions:

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**.

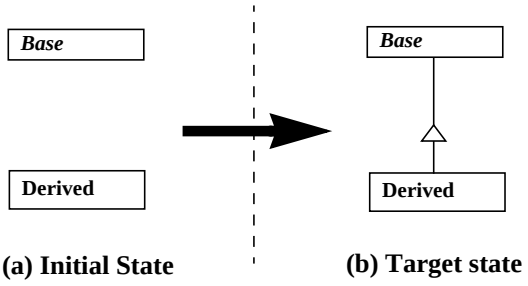


Figure 2.1: Inherit[*Base*, Derived] transformation

3.2 Preserving behavior

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

Opdyke identifies seven invariants required to preserve the behavior of C++ and Smalltalk programs [Opd92]:

1. **Unique superclass**. After refactoring, a class must always have at most one direct superclass and its superclass must not also be one of its subclasses. This limits the focus of this research to single inheritance systems.
2. **Distinct class names**. After refactoring, each class must have a unique class name and classes cannot be nested.
3. **Distinct member names**. After refactoring, all member variables and functions within a class must have distinct names. This does allow for a member function in a superclass to be overridden in a subclass.
4. **Inherited member variables not redefined**. A member variable inherited from a superclass cannot be redefined in any of its subclasses.
5. **Compatible signatures in member function redefinition**. After refactoring, if a member function defined in a superclass is redefined in a subclass, it must maintain the same type signature.
6. **Type-safe assignments**. After refactorings, the type of each expression assigned to a variable must be an instance of the variable's defined type, or (if it is a pointer variable) possibly an instance of one of its subtypes.
7. **Semantically equivalent references and operations**. This condition

allows for simplification of expressions, removal of dead code, addition of variables and functions if they are unreferenced, changing the type of a variable as long as each operation on the variable is defined equivalently for the new type, and replacing the references to a variable or function in one class with equivalent references to a variable or function defined in another class. A complete discussion of this invariant is given in Opdyke's thesis [Opd92, pp. 28-30].

All refactorings in our work preserve these invariants. For the **inherit** example (Figure 2.1):

1. The first enabling condition states that ***Base*** must not be a subclass of **Derived** and **Derived** must not have a superclass. This preserves the unique superclass invariant.
2. **Inherit** does not add or change any class names preserving the distinct class names invariant.
3. **Inherit** does not add or change any member variable or function names preserving the distinct member names invariant.
4. The second enabling condition states that member variables of **Derived** must have distinct names from member variables of ***Base*** and its superclasses. This preserves the inherited member variables not redefined invariant.
5. The third enabling condition states that a member function of **Derived** which overrides a function must have the same type signature as the function it overrides. This preserves the compatible signatures in member function redefinition invariant.
6. **Inherit** does not change the type of any expression assigned to a variable preserving the type-safe assignments invariant.
7. Finally, **inherit** does not change any variables referenced or functions

invoked preserving the semantically equivalent references and operations invariant. Note that **inherit** only creates an inheritance relationship but it does not perform any kind of substitution.

The claim is that preservation of these invariants is proof of behavior preservation for C++ and Smalltalk. However, our experiments and analysis show that while the invariants may preserve behavior for object-oriented databases or specification-to-source transformation systems, they did not preserve behavior for source-to-source transformation systems because of complexities introduced by the language being transformed. When a refactoring was found to change behavior, we created an invariant to be preserved. We contribute the following additional invariants:

8. **Implementation of pure-virtual functions**. If a class is instantiated, then it cannot have any pure-virtual functions. C++ does not allow an instantiated class to inherit pure-virtual functions — the functions must be overridden [Ell90][1].
9. **Maintaining aggregate objects**. If a program depends on the aggregate property of an object, then that property must be preserved. An object of a class is an *aggregate* if that class has no constructors, no private or protected members, no superclasses, and no virtual functions. In C++, aggregates are the only objects which can be initialized by initializer lists [Ell90][2].
10. **No instantiation side-effects**. If a refactoring can change the number of times a class is instantiated, then the constructor cannot have any side-

1. Opdyke's **add_variable** refactoring allows you to add an instance variable whose type is a class which declares or inherits a pure-virtual function.
2. An example violation of this invariant was discovered in our experiments described in Section 6.4.2.

effects beyond initializing the object created[3].

The **inherit** refactoring preserves these invariants:

- The fourth enabling condition states that subclasses of ***Base*** must implement any pure virtual methods if objects of that class are created. This preserves the implementation of pure-virtual functions invariant.
- The fifth enabling condition states that initializer lists must not be used to initialize **Derived** objects. In C++, the only special property of aggregates is that they can be initialized with initializer lists. This condition preserves the maintaining aggregate objects invariant.
- The sixth enabling condition states that 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. This preserves the no instantiation side-effects invariant.

Finally, Opdyke states that his refactorings do not apply to programs which are dependent on the size or physical layout of objects. Unlike the multiple inheritance condition which is prevented by the unique superclasses invariant, Opdyke chose not to create an invariant to govern this condition. For consistency, we create an invariant to handle this case:

11. **Size and layout independence**. If a program is dependent of the size and layout of objects, it cannot alter their size or layout.

The seventh enabling condition of **inherit** preserves this invariant. We make no claim that this expanded list of invariants is complete. A formal proof of behavior preservation would require a rigorous analysis of the ANSI specification which is

3. For example, if a class maintains a class variable of the number of objects created and its constructor prints this number, the behavior of a program would change if the number of class instantiations changed. Opdyke's **add_variable** refactoring allows you to add to class **C** an instance variable whose type is class **H**. After refactoring, code which instantiates class **C** will also instantiate class **H** changing the number of times **H** is instantiated.

beyond the scope of this dissertation. Rather, we treat invariants as test cases which must be satisfied for each refactoring. Having more test cases increases the reliability of refactorings, but does not guarantee their freedom from errors. Our views on behavior preservation are discussed further in Section 8.2.

3.3 Conservatism of enabling conditions

A *conservative condition* refers to a condition which is more restrictive than necessary to preserve behavior. Conservative conditions are easier to design and implement.

While a reduction in conservatism leads to more powerful refactorings, it is generally unrealistic to seek a non-conservative refactoring implementation. Consider the non-conservative check of enabling conditions for the semantically equivalent references and operations invariant in the case where a type is changed. A type change requires that operations performed on a variable of the old type are equivalent to operations performed on a variable of the new type. Since operations are C++ routines, the non-conservative solution to this problem requires you to determine whether two C++ routines are semantically equivalent where semantic equivalence is defined as having the same inputs map to the same outputs. One can imagine two arbitrarily complex routines whose semantic equivalence would be virtually impossible to detect for both computers and humans.

3.4 Verification of enabling conditions

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

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

While it may be possible to introduce conservative enabling conditions to determine that a program's behavior is independent of object size (e.g. restrict recasting, disallow pointer arithmetic, and disallow `sizeof`, etc.), the resulting refactoring may have limited applicability because of the conservatism. The **inherit** refactoring requires that programs be independent of object size since adding a superclass can change the size of an object. Thus, not all enabling conditions for **inherit** are checked automatically.

3.5 List of refactorings

The list of refactorings used in our research is given in Table 1. In addition to the refactorings proposed by Banerjee and Kim for evolving object-oriented database schemas [Ban87] and by Opdyke for restructuring object-oriented programs [Opd92], we found that transforming actual C++ programs required the following new refactorings:

- We enlarged the set of schema evolutions to include, for example, **inherit** (Figure 2.1) and **substitute**. **Substitute** changes a class's dependency on a class **C** to a dependency on a superclass of **C** [Tok95].

- Other refactorings are language-specific; **procedure_to_method** and **structure_to_class** convert C artifacts to their C++ equivalents.
- A third set of refactorings supports the addition of patterns in evolving programs [Tok95, Tok99]. Examples include **add_factory_method**, **singleton**, and **procedure_to_command**. **Add_factory_method** creates a method which returns a new object, **singleton** creates a class with only one instance, and **procedure_to_command** converts a C procedure to a singleton class with a method for executing the procedure.

The refactorings added to the lists of Banerjee, Kim, and Opdyke are italicized in Table 1. A complete definition of each refactoring is provided in Appendix B.

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

Table 1: Refactorings

Chapter 4

Evolving designs with refactorings

Methods for evolving designs appear to follow regular patterns, particularly for object-oriented applications. Three kinds of object-oriented design evolution that have been identified to date are: schema transformations, design pattern microarchitectures, and hot-spot meta patterns.

- *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]. Designs using abstract classes and template methods are prescribed to keep these hot-spots flexible.

This chapter catalogs the schema transformations, design patterns, and hot-spot meta patterns whose introduction can be automated with refactorings. Refactoring support for design patterns and hot-spot meta patterns is an original contribution of this research.

4.1 Object-oriented schema transformations

A database schema for an object-oriented database management system (OODBMS) looks like a class diagram for an object-oriented application. Consequently, 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 4.1). This particular 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 12 have been implemented as automated refactorings[1] (see Table 4.1). We also implement three additional schema transformations not

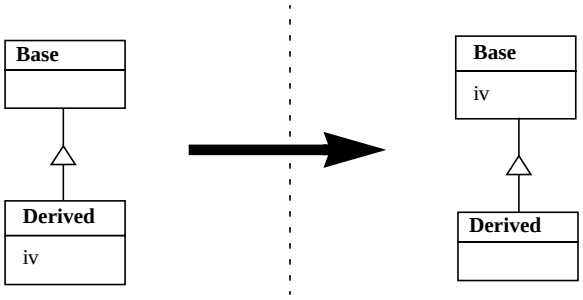


Figure 4.1: Using pull_up_variable to move instance variable "iv" from Derived to Base

1. The seven refactorings which are not supported are: a) changing the value of a class variable, b) changing the code of a method, c) changing the default value of an instance variable, d) changing the inheritance parent of an instance variable, e) changing the inheritance of a method, f) adding a method, and g) 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 f) cannot be automated. Finally, g) is not supported because this research is currently limited to applications without multiple inheritance.

Description from [Ban87]	Refactoring
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**

Table 4.1: Schema changes supported as refactorings

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.

Description	Refactoring
Move a method through a composite link	**move_method_across_object_boundary**
Move a variable through a composite link	**move_variable_across_object_boundary** (Figure 4.2)
Change a class' dependency on a class **C** to a dependency on a superclass **S** of **C**	**substitute** (Figure 4.3)

listed in [Ban87] (Table 4.2). **Move_method_across_object_boundary** was proposed by Roberts [Rob97] while we proposed the latter two refactorings [Tok95, Tok99a]. Figure 4.2 and Figure 4.3 display the

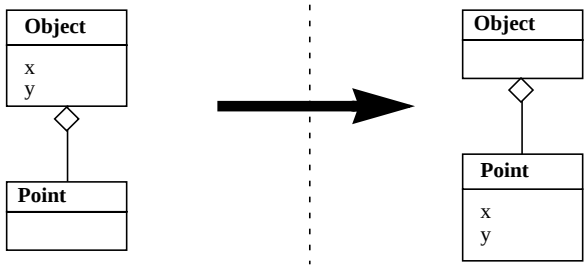


Figure 4.2: Using move_variable_across_object_boundary to move instance variables x and y

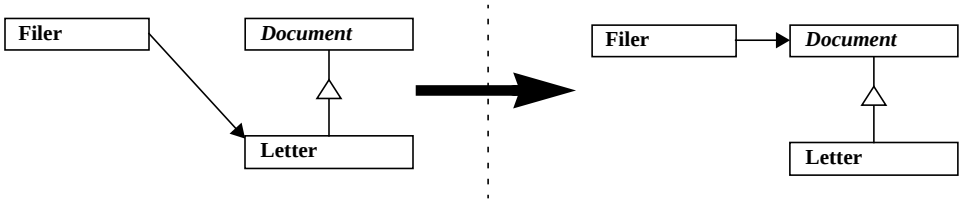


Figure 4.3: Using substitute to change Filer's reference to a Letter to a reference to a Document

move_variable_across_object_boundary and **substitute** refactorings when viewed at the level of a class diagram (see Appendix A for a complete definition).

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. Opdyke first proposed applying a series of schema transformations to create abstract superclasses and capture components [Opd92].

4.2 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. Six patterns are automatable as refactorings (see Table 4.3).

Pattern	Description
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. (See example in Section 6.4.1.)
Composite	Composite composes objects into tree structures to represent part-whole hierarchies. The **composite** refactoring converts a class into a composite class. (See example in Section 4.3.2.)
Decorator	Decorator attaches additional responsibilities to an object dynamically. The **decorator** refactoring converts a class into a decorator class. (See example in Section 4.3.2.)
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. (See example in Section 5.2.)
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.
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. (See example in Appendix A.)

Table 4.3: Design patterns supported as refactorings

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 [Tok95]. At least seven patterns from [Gam95] can be viewed as a program transformation

(see Table 4.4).

In all cases, we can apply refactorings to simple designs to create the designs

Pattern	Description	Example
Abstract Factory	Abstract Factory provides an interface for creating families of related or dependent objects without specifying their concrete class.	Section 5.2
Adapter	Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.	Section 4.2.1
Bridge	Bridge decouples an abstraction from its implementation so that the two can vary independently.	[Sch98][a], Section 4.2.2
Builder	Builder separates the construction of a complex object from its representation so that the same construction process can create different representations.	Appendix B
Strategy	Strategy lets algorithms vary independently from the clients that use them.	Appendix B
Template Method	Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure.	Section 4.3.2
Visitor	Visitor lets you define a new operation without changing the classes of the elements on which it operates.	[Rob97]

Table 4.4: Design patterns supported as a composition of refactorings

a. The example from [Sch98] is only partially automated because they were limited to the original refactorings from [Opd93]. A fully automated example is provided in Section 4.2.2.

used as prototypical examples in [Gam95]. The following sections show how the Adapter and Bridge design patterns can be automated. Derivations for other patterns are provided in Appendix B.

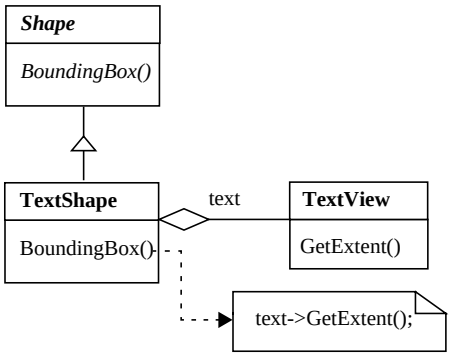


Figure 4.4: TextShape adapts TextView's interface

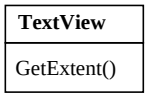


Figure 4.5: Unadapted TextView class

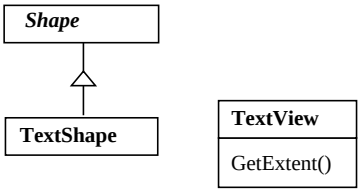


Figure 4.6: Adapter class created

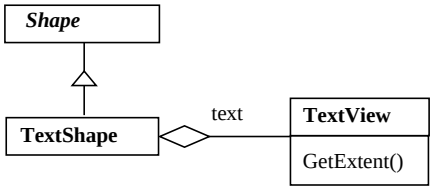


Figure 4.7: Adaptee instance variable added to adapter

4.2.1 Adapter

Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. In the object adapter example from [Gam95] (Figure 4.4), the **TextShape** class adapts **TextView**'s `GetExtent()` method to implement `BoundingBox()`. The adapter can be constructed from the original **TextView** class (Figure 4.5) in four steps:

1. *Create the adapting classes.* Create the classes **TextShape** and ***Shape*** using **create_class**. Make **TextShape** a subclass of ***Shape*** using **inherit** (Figure 4.6).
2. *Add an instance variable holding an object to be adapted.* Add the `text` instance variable to **TextShape** using **add_variable** (Figure 4.7).

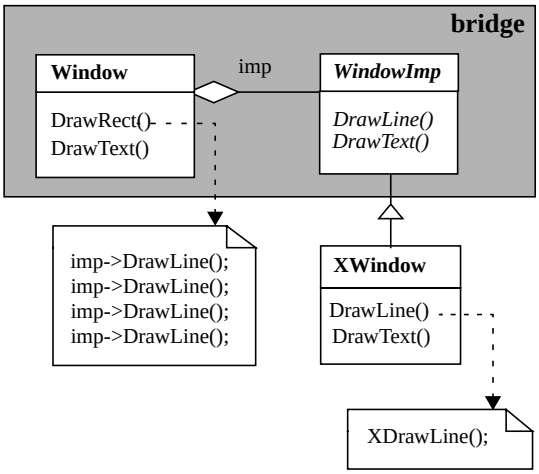


Figure 4.8: Bridge design pattern example

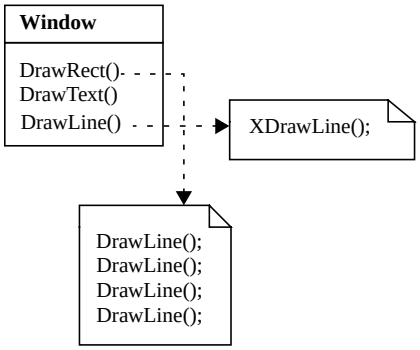


Figure 4.9: Design for a single window system

3. *Add adapted methods to the adapter's interface.* Create the `BoundingBox()` method which calls `text->GetExtent()` using **create_method_accessor**. **Create_method_accessor** creates a method which replaces calls of the form `instance_variable->method()`.
4. *Declare the methods in the adapter's superclass.* Declare `BoundingBox()` in ***Shape*** using **declare_virtual_method** (Figure 4.4).

4.2.2 Bridge

Bridge decouples an abstraction from its implementation so that the two can vary independently. In the example from [Gam95] (Figure 4.8), 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 ***Window-Imp*** 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.

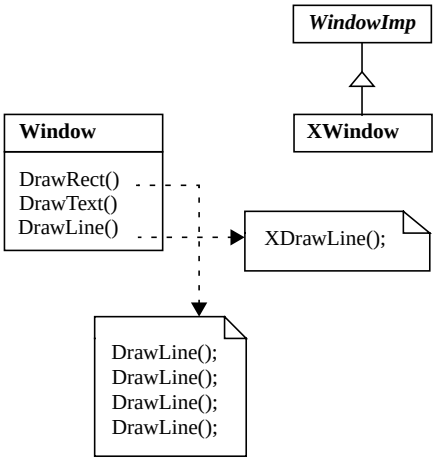


Figure 4.10: Implementor classes created

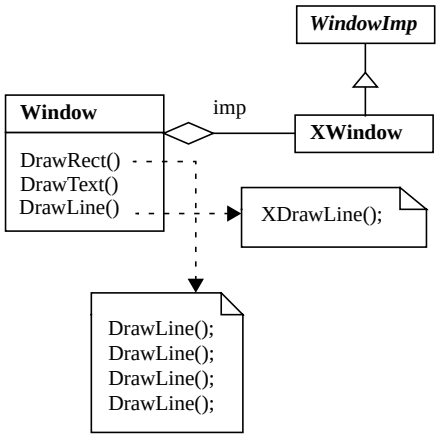


Figure 4.11: Implementor instance variable added to Window

Refactorings can be used to install a bridge design pattern given a simple design committed to a single window system. Figure 4.9 depicts a system designed for X-Windows. This system can be evolved with refactorings to use the bridge design pattern in five steps:

1. *Create the implementor classes.* Create classes **XWindow** and ***WindowImp*** using **create_class**. Make ***WindowImp*** a superclass of **XWindow** with **inherit** (Figure 4.10).
2. *Add an instance variable which links the abstraction to an implementation.* Add instance variable `imp` to the **Window** class using **add_variable** (Figure 4.11).
3. *Move methods required for an implementation to the implementor class.* 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. Add a `DrawText()` method to **Window** which calls `DrawText()` in ***WindowImp*** using **create_method_accessor**. This preserves **Window**'s interface[2] (Figure 4.12).

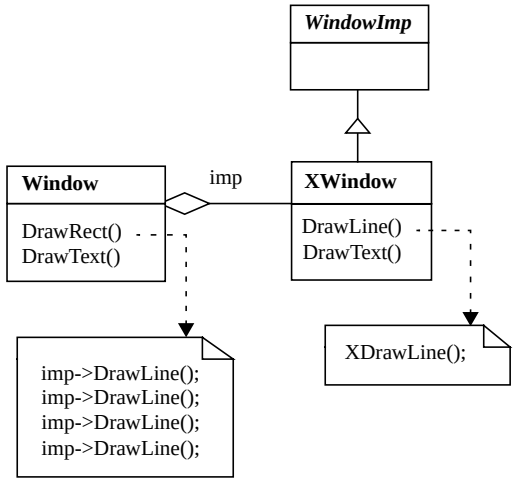


Figure 4.12: Window system specific methods moved to XWindow class

4. *Declare the methods in the implementor superclass.* Declare method `DrawLine()` and `DrawText()` in ***WindowImp*** with **declare_virtual_method** (Figure 4.13.)
5. *Generalize the abstraction to accept any implementation.* Change the type of instance variable `imp` from **XWindow** to ***WindowImp*** using **substitute** (Figure 4.8).

The Bridge microarchitecture uses object composition to provide needed flexibility. Object composition is also present in the Builder and Strategy design patterns (see Appendix B). 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.

2. Note that the original Window class also had a DrawLine() method which was used to implement DrawRect(). The target design from [Gam95] did not include DrawLine() as a part of Window's interface, thus, no accessor for it was added in this example.

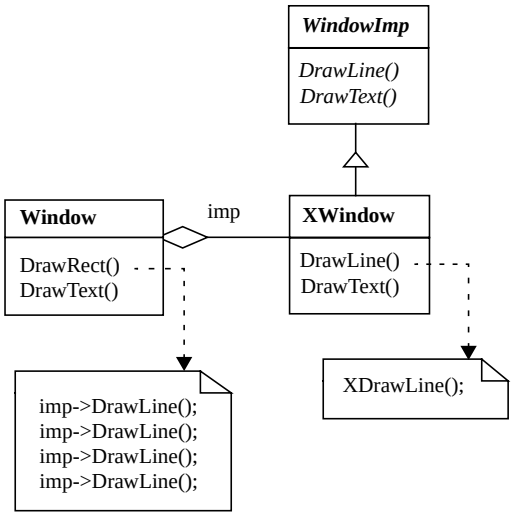


Figure 4.13: Window system specific methods moved to XWindow class

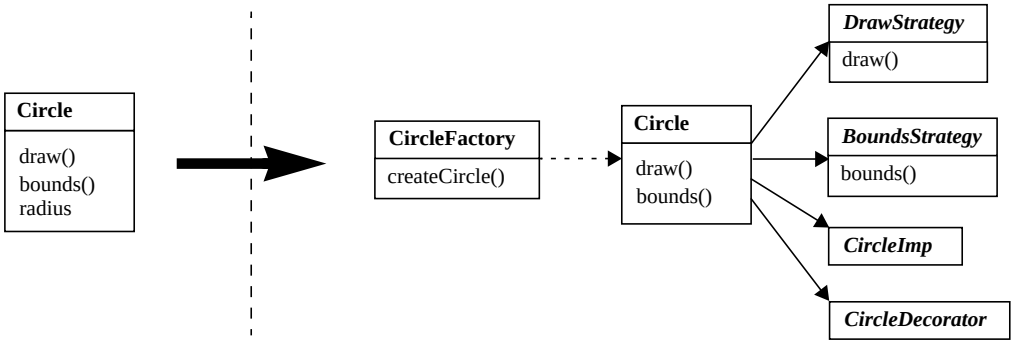


Figure 4.14: Overenthusiastic use of design patterns

4.2.3 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 4.14. 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 than what is actually required. The migration from a single **Circle** class to the complex microarchitecture in Figure 4.14 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 to be *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.

4.3 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. A Circle factory is created [Tok95]. Strategies are added (Section 4.2). The Bridge pattern is applied (Section 4.2). Finally, a decorator is added (Section 4.2).

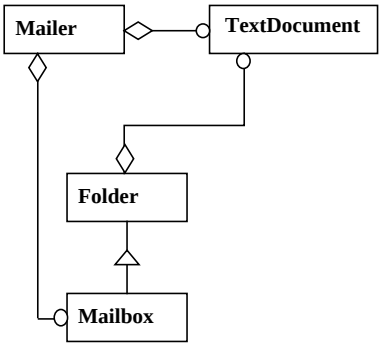


Figure 4.15: Initial state of mailing system

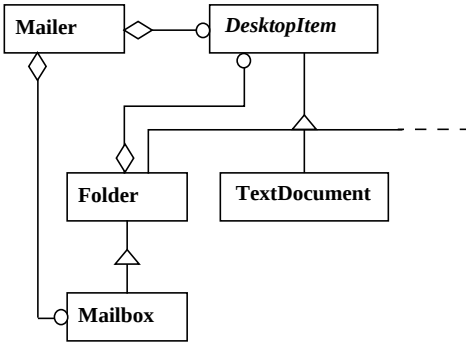


Figure 4.16: Final state of mailing system

4.3.1 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 4.15 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 4.16. Under the improved design, **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 4.17).
2. Make ***DesktopItem*** a superclass of **TextDocument** using **inherit** (Figure 4.18).
3. Generalize the link between **Mailer** and **TextDocument** to a link between **Mailer** and ***DesktopItem*** using **substitute** (Figure 4.19). Subclasses of ***DesktopItem*** can now be mailed.
4. Generalize the link between **Folder** and **TextDocument** to a link between **Folder** and ***DesktopItem*** using **substitute** (Figure 4.20). **Folder** can now contain any ***DesktopItem***.

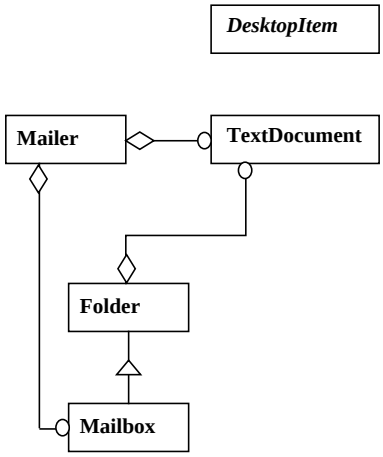


Figure 4.17: Empty TextDocument class created

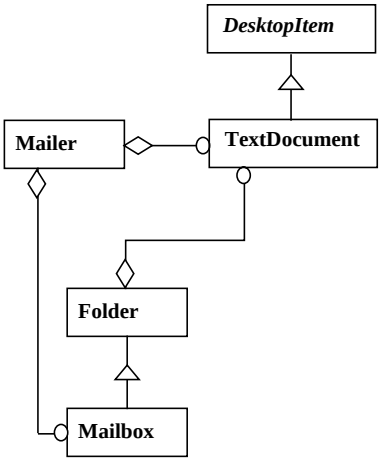


Figure 4.18: TextDocument inherits from *DesktopItem*

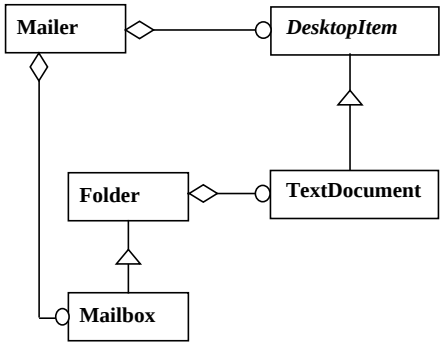


Figure 4.19: Mailer dependency changed from TextDocument to *DesktopItem*

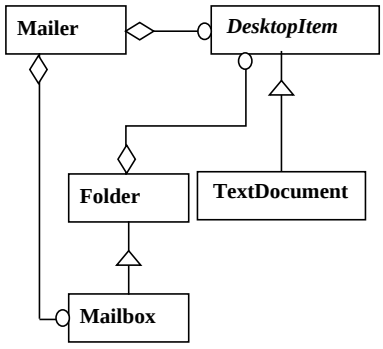


Figure 4.20: Folder can contain any *DesktopItem*

5. Make **Folder** a subclass of ***DesktopItem*** using **inherit** (Figure 4.16). A **Folder** which can contain a ***DesktopItem*** can now contain another **Folder**.

With the improved design, 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.

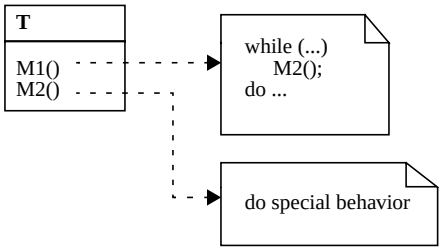


Figure 4.21: Template and hook methods in same class

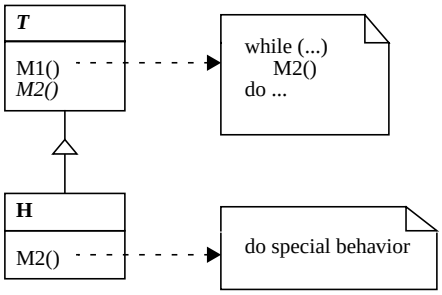


Figure 4.22: Hook method M2() overridden in class H

4.3.2 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 4.21 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 4.22). 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 can be applied to a design to add meta patterns or to transition from one meta pattern to another. The transitions between patterns enabled by refactorings are displayed in Figure 4.23.

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 (Figure 4.24). 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. The

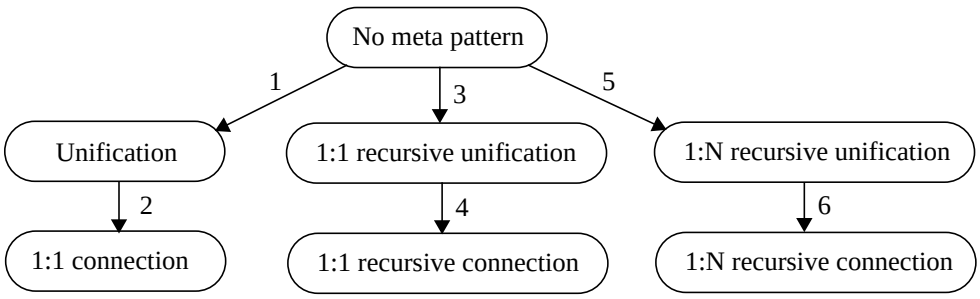


Figure 4.23: Hot-spot meta pattern transitions enabled by refactorings

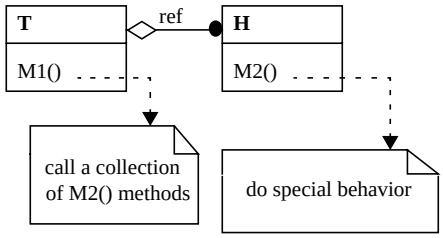


Figure 4.24: 1:N connection meta pattern

remainder of this section demonstrates automation of the first two transitions in Figure 4.23. Derivations for the remaining transitions are given in Appendix B.

Figure 4.25 displays the transformation of a design with no template or hook methods to a design using the unification meta pattern for which both template and hook methods are located in the same class (transition 1 from Figure 4.23). Class `T` having method `M1()` which calls some special behavior. A hook method can be added with refactorings in one step:

1. *Encapsulate the special behavior as a method.* Create a hook method `M2()` which executes the special behavior using **extract_code_as_method** (Figure 4.25, right hand side).

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

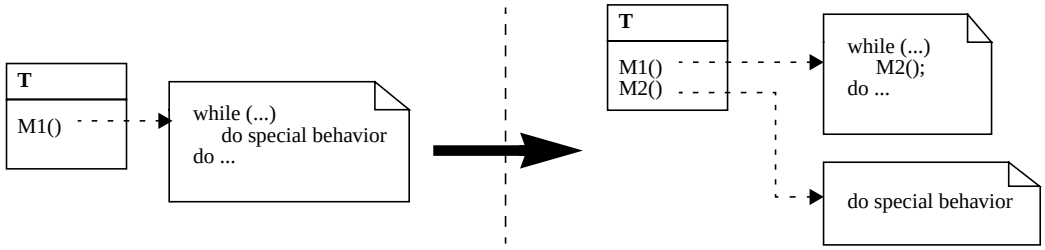


Figure 4.25: Method M1() calls a special behavior which differs for each application

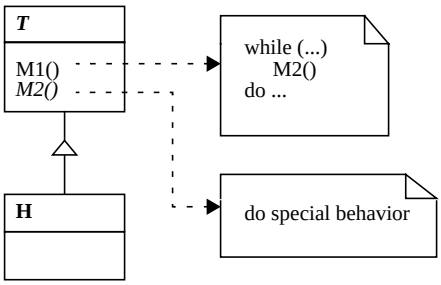


Figure 4.26: Hook class created

`M1()`. The extended structure can be added in three steps:

1. *Create a hook class as a subclass of the template class.* Create class **H** using **create_class**. Make **H** a subclass of ***T*** using **inherit** (Figure 4.26).
2. *Declare the hook method in the template class so it can be overridden in the hook class.* Declare `M2()` in ***T*** using **declare_virtual_method**.
3. *Move the hook method implementation to the hook class.* Move `M2()` into **H** using **push_down_method** (Figure 4.22).

Figure 4.27 displays the transformation from unification to 1:1 connection (transition 2 from Figure 4.23). Consider the 1:1 connection meta pattern which stores the hook method in an object owned by the template class (Figure 4.27, right hand side). 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 4.27, left hand side) as a starting

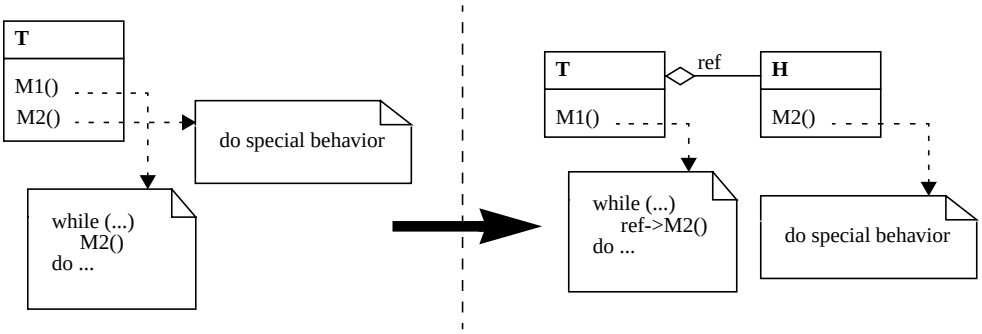


Figure 4.27: Unification to 1:1 connection

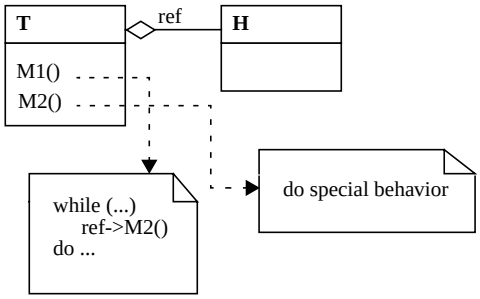


Figure 4.28: Connection to H object created

point.

1. *Create the hook class.* Create class **H** using **create_class**.
2. *Link the template class to the hook class.* Add an instance variable `ref` to **T** with **add_variable** (Figure 4.28).
3. *Move the hook method to the hook class.* Move `M2()` to class **H** using **move_method_across_object_boundary** (Figure 4.27, right hand side).

The behavior of template method `M1()` can now be altered dynamically by pointing to different hook class objects with different implementations of `M2()`.

4.3.3 Role of refactorings for hot-spot analysis

The hot-spot-driven-approach provides a comprehensive method for evolving designs to manage changes 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. This section demonstrates that 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.

4.4 Role of refactorings for object-oriented design evolution

Design 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 designs evolve. Three modes of design 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 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

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 Chapter 6 where many hand-coded changes between two major releases are automated.

Chapter 5

Evolving an application

This chapter demonstrates the role of refactorings in evolving object-oriented applications. Refactorings can automate design evolution to produce new designs which are more extensible and reusable.

5.1 Refactorings and software evolution

Refactorings transform a software system from one compilable state to another compilable state while preserving behavior. For this reason, they are well suited to the principled development style advocated by O'Shea, Beck, Casais and others. This style performs improvements by first transforming the design while preserving program behavior, and then extends the better designed system [OSh86, Cas91]. Design transformation is currently a manual process. Refactorings can automate some or all of this process (Figure 5.1).

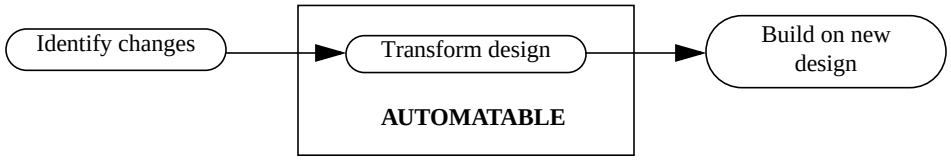


Figure 5.1: Software evolution processes

The steps involved in transforming a design include: identifying source code

affected by a design change, implementing the change, testing the change, fixing bugs, and retesting the application until the risk of introducing new errors is minimized. Refactorings automate the identification of source code affected by a change and the execute the changes. The behavior preserving nature of refactorings eliminates the need for a test and debug cycle[1]. The example presented in this chapter demonstrates that refactorings can automate design changes which have a widespread impact on both an application's design and on its underlying source code.

5.2 A refactoring example

We use a proof-of-concept example to illustrate the design evolution process. The example evolves a Car Factory application that creates a Honda Prelude with a **VTEC2_2** engine and **GY184_HR14** tires. The Prelude is driven for one million miles during which time tires are rotated and changed, and the engine is replaced. The program and its class diagram are displayed in Figure 5.2a and Figure 5.3.

Inspection of the class diagram reveals that the application will create one kind of car with a single choice for an engine and tires. At some point, we want to create other cars with different engines and tires. There are several problems that must be addressed to generalize this program. In this example, we use object-oriented transformations to create and install an abstract factory and demonstrate that the resulting program is easier to extend and reuse. The design is transformed in six steps:

1. *Create superclasses to support other tires and engines.* ***Tire*** and ***Engine*** classes are created using **create_class**. ***Tire*** and ***Engine*** are then declared to be superclasses of **GY184_HR14** and **VTEC2_2** respectively using **inherit**.

1. See Section 8.2 for a detailed discussion on behavior preservation.

```
#include <iostream.h>

class GY184_HR14 {
public:
  int miles, max_miles;
  void Drive(int m) {
    miles += m;
  }
  GY184_HR14 () {
    max_miles = 40000;
    miles = 0;
  }
};
class VTEC2_2 {
public:
  int miles, max_miles;
  void Drive(int m) {
    miles += m;
  }
  VTEC2_2 () {
    max_miles = 100000;
    miles = 0;
  }
};
class Car {
private:
  GY184_HR14 *lf_tire, *lb_tire,
    *rf_tire,*rb_tire;
  VTEC2_2 *engine;

public:
  int miles;
  Car();
  void RotateTires() {
    GY184_HR14 *tire;

    tire = lf_tire;
    lf_tire = lb_tire;
    lb_tire = lf_tire;
    tire = rf_tire;
    rf_tire = rb_tire;
    rb_tire = rf_tire;
    cout << "Rotating tires\n";
  }
  void ReplaceTires(GY184_HR14 *lf,
      GY184_HR14 *rf, GY184_HR14 *lb,
      GY184_HR14 *rb) {
    delete lf_tire;
    delete lb_tire;
    delete rf_tire;
    delete rb_tire;
    lf_tire = lf;
    lb_tire = lb;
    rf_tire = rf;
    rb_tire = rb;
    cout << "Replacing tires\n";
  }
  void ReplaceEngine(VTEC2_2 *e) {
    delete engine;
    engine = e;
    cout << "Replacing engine\n";
  }
  void Drive(int m) {
    miles += m;
    engine.Drive(m);
    lf_tire.Drive(m);
    lb_tire.Drive(m);
    rf_tire.Drive(m);
    rb_tire.Drive(m);
    cout << "Driving " << m << " miles\n";
  }
  int TireMiles() {
    return lf_tire->miles;
  }
  int MaxTireMiles() {
    return lf_tire->max_miles;
  }
  int EngineMiles() {
    return engine->miles;
  }
  int MaxEngineMiles() {
    return engine->max_miles;
  }
};

class App {
public:
  App() {
  }
  void run();
};

App *app;

Car::Car(GY184_HR14 *lf,
     GY184_HR14 *lb,
     GY184_HR14 *rf,
     GY184_HR14 *rb, VTEC2_2 *e) {
  engine = new VTEC2_2;
  lf_tire = new GY184_HR14;
  lb_tire = new GY184_HR14;
  rf_tire = new GY184_HR14;
  rb_tire = new GY184_HR14;
}
void App::run() {
  Car *car = new Car(new GY184_HR14,
      new GY184_HR14, new GY184_HR14,
      new GY184_HR14, new VTEC2_2);
  while (car->miles < 1000000) {
    car->Drive(1000);
    if (car->TireMiles() >=
        car->MaxTireMiles())
      car->ReplaceTires(new GY184_HR14,
        new GY184_HR14, new GY184_HR14,
        new GY184_HR14);
    else if (car->TireMiles() %5000 ==0)
      car->RotateTires();
    if (car->EngineMiles() >=
        car->MaxEngineMiles())
      car->ReplaceEngine(new VTEC2_2);
  }
  cout << "Total miles driven: " <<
    car->miles << "\n";
}

int main () {
  app = new App;
  app->run();
}
```

Figure 5.2a: Car Factory program

```
class Tire {   // Tire class created
};

class Engine { // Engine class created
};

// subclass of Tire
class GY184_HR14 : public Tire {
public:
int miles, max_miles;
  void Drive(int m) {
    miles += m;
  }
};

// subclass of Engine
class VTEC2_2 : public Engine {
public:
  int miles, max_miles;
  void Drive(int m) {
    miles += m;
  }
};
```

Figure 5.2b: Creating Tire and Engine superclasses

```
class Tire {
public:
  int miles, max_miles; // from subclass
  void Drive(int m) {   // from subclass
    miles += m;
  }
};

class Engine {
public:
  int miles, max_miles; // from subclass
  void Drive(int m) {   // from subclass
    miles += m;
  }
};

class GY184_HR14 : public Tire {
};

class VTEC2_2 : public Engine {
}
```

Figure 5.2c: Moving variables and methods to superclasses

```
class Car {
public:
  // variables changed from GY184_HR14 to
Tire
  Tire *lf_tire, *lb_tire, *rf_tire,
    *rb_tire;

  // variable changed from VTEC2_2 to
Engine
  Engine *engine;

  void RotateTires() {
    // variable type changed from
GY184_HR14
    // to Tire
    Tire *tire;
    ...
  }

  // variables changed from GY184_HR14 to
Tire
  void ReplaceTires(Tire *lf, Tire *rf,
                    Tire *lb, Tire *rb) {
    ...
  }

  // variable type changed from VTEC2_2
  // to Engine
  void ReplaceEngine(Engine *e) {
    ...
  }

  ...
};
```

Figure 5.2d: Generalizing Car class

```
class PreludeFactory {
public:
  // new method to create Tires
  Tire *MakeTire() {
    return new GY184_HR14;
  }


  // new method to create Engine
  Engine *MakeEngine() {
    return new VTEC2_2;
  }
};

class Car {
public:
  ...
  // arguments to constructor generalized
  Car (Tire *, Tire *,
    Tire *, Tire *, Engine *);
  ...
}

class App {
public:
  Car *car;

  // new instance variable created
  PreludeFactory *car_factory;

  App() {
    // new instance variable initialized
    car_factory = new PreludeFactory;

    car = new Car;
  }
  void run();
};
```

```
Car::Car(Tire *, Tire *,
  Tire *, Tire *, Engine *) {
  miles = 0;

    // use factory method to create engine
     engine = app->car_factory
         ->MakeEngine();

  // use factory method to create tires
  lf_tire = app->car_factory->MakeTire();
  lb_tire = app->car_factory->MakeTire();
  rf_tire = app->car_factory->MakeTire();
  rb_tire = app->car_factory->MakeTire();
}

void App::run() {
  // use factory methods to create tires
  // and engine
  Car *car = new Car(
    app->car_factory->MakeTire(),
    app->car_factory->MakeTire(),
    app->car_factory->MakeTire(),
    app->car_factory->MakeTire(),
    app->car_factory->MakeEngine());
  while (car->miles < 1000000) {
    car->Drive(1000);

    // use factory method to create tires
    if (car->TireMiles() >=
        car->MaxTireMiles())
      car->ReplaceTires(
        app->car_factory->MakeTire(),
        app->car_factory->MakeTire(),
        app->car_factory->MakeTire(),
        app->car_factory->MakeTire());
    else if (car->TireMiles() %5000 == 0)
      car->RotateTires();

    // use factory method to create the
    // engine
    if (car->EngineMiles >=
        car->MaxEngineMiles)
      car->ReplaceEngine(
        car_factory->MakeEngine());
  }
  cout << "Total miles driven: " <<
    car->miles << "\n";
}
```

Figure 5.2e: Add concrete factory

```
class CarFactory {
public:
  // virtual method added
  virtual Tire *MakeTire();

  // virtual method added
  virtual Engine *MakeEngine();
};

// PreludeFactory is now a subclass of
// CarFactory
class PreludeFactory : public CarFac-
tory {
  ...
};
```

Figure 5.2f: Create abstract factory

```
class App {
public:
  ...
  // instance variable changed from
  // PreludeFactory to CarFactory
  CarFactory *car_factory;
  ...
}
```

Figure 5.2g: Generalize App class

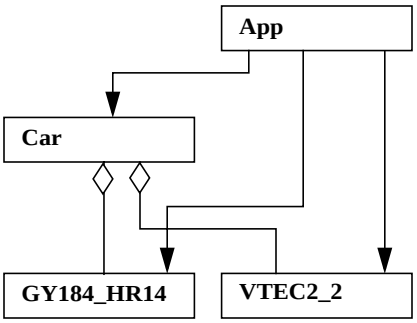


Figure 5.3: Initial class diagram

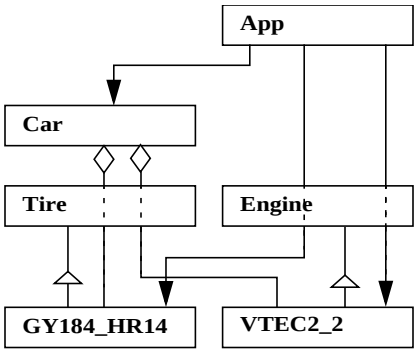


Figure 5.4: Tire and Engine superclasses created

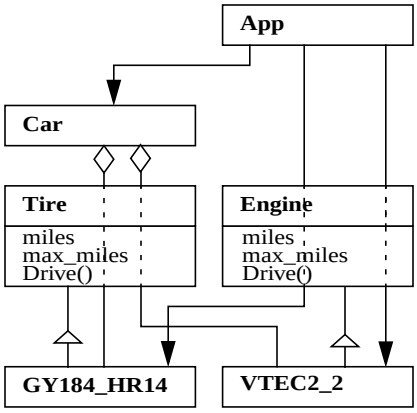


Figure 5.5: : Variables and methods moved to superclass

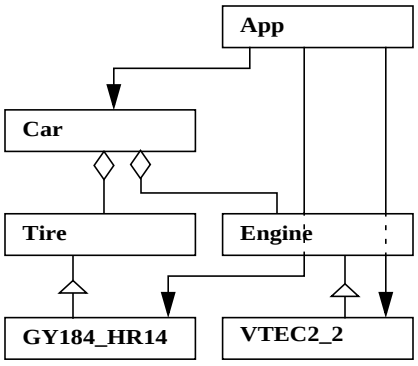


Figure 5.6: Car dependency on GY184_HR14 and VTEC2_2 is removed

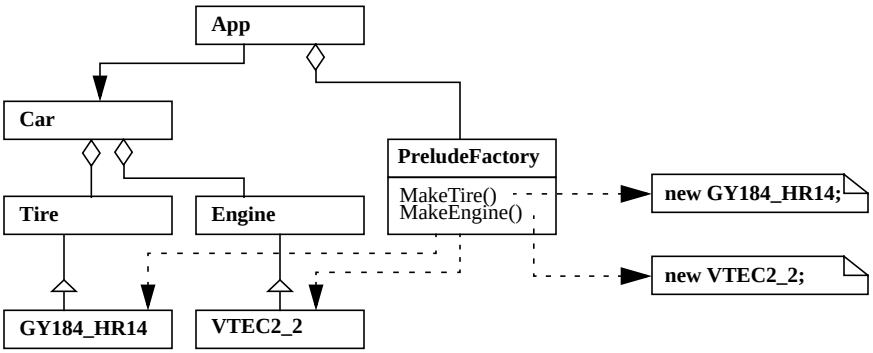


Figure 5.7: Concrete factory added

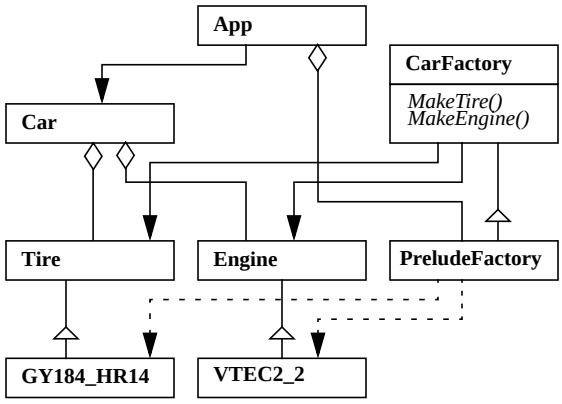


Figure 5.8: Abstract factory created

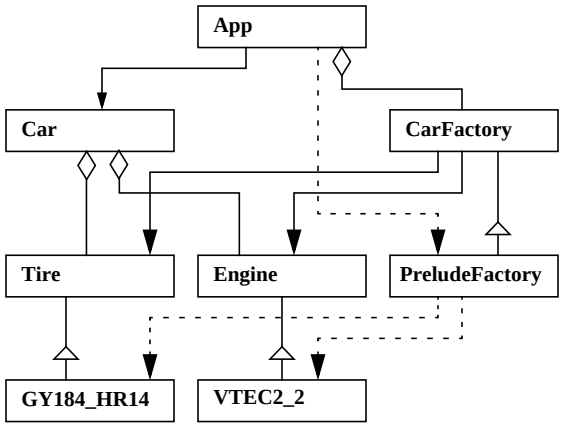


Figure 5.9: App reference to prelude factory is removed.
App still creates a PreludeFactory object.

The resulting code changes and class diagram are displayed in Figure 5.2b and Figure 5.4.

2. *Move variables and methods to the superclasses.* We can now move variables and methods common to all tires and engines from the **GY184_HR14** and **VTEC2_2** classes to the ***Tire*** and ***Engine*** classes using **pull_up_variable** and **pull_up_method** (Figure 5.2c and Figure 5.5).

Classes derived from ***Tire*** and ***Engine*** can now inherit state and behavior originally belonging to the **GY184_HR14** and **VTEC2_2** classes.

3. *Generalize the* **Car** *class to use any* ***Tire*** *or* ***Engine.*** We take advantage of the new ***Tire*** and ***Engine*** superclasses by substituting them for **GY184_HR14** and **VTEC2_2** respectively using the **substitute** refactoring. The ***Car*** class can then operate with any classes derived from ***Tire*** and ***Engine*** (Figure 5.2d and Figure 5.6).

   Note that this transformation can only be applied if the enabling conditions are satisfied. In this example, the **Car** class must not reference any subclass specific instance variables or methods from **GY184_HR14** or **VTEC2_2**. We know this to be true because we moved the instance variables and methods referenced by **Car** to the ***Tire*** and ***Engine*** classes in Step 2.

4. *Add a concrete factory for creating the tires and engines used to construct Honda Preludes.* **add_variable** is called to add an instance variable to **App** which stores the concrete factory to be used for constructing all parts (in this case **PreludeFactory**). Factory methods are added to **PreludeFactory** to create the appropriate tires and engines using **add_factory_method**. The resulting code changes and class diagram is displayed in Figure 5.2e and Figure 5.7. The concrete factory guarantees that the correct tires and engine will be created when constructing a Prelude.

5. *Create an abstract factory.* Next we create the abstract factory ***CarFactory*** as a superclass of the concrete factory **PreludeFactory** (Figure 5.2f and Figure 5.8). ***CarFactory*** is used as a base class for deriving concrete factories to produce other types of cars.

6. *Generalize* **App** *to use* ***CarFactory.*** Using **substitute**, we generalize the **App** class to change its dependency on ***PreludeFactory*** to a dependency on

CarFactory (Figure 5.2g and Figure 5.9). **App** will now run for any type of car. Note that in Step 4, we added an instance variable to **App** which points to a **PreludeFactory** instance. We will later show that the type of car created and driven by **App** can be changed by pointing to a different concrete factory.

The program now contains the abstract factory **CarFactory** with concrete factory **PreludeFactory**. ***CarFactory*** objects produce objects of the ***Tire*** and ***Engine*** classes. **PreludeFactory** produces the **GY184_HR14** and **VTEC2_2** objects required by a Prelude. The original program was 136 lines and the final version was 171 lines. 65 lines of code were added or changed through automated refactorings.

5.3 Benefits of the refactored system

The transformed program now employs the Abstract Factory design pattern which guarantees that only coordinated car components will be produced. A comparison of the initial (Figure 5.3) and final (Figure 5.9) class diagrams reveals that the transformed program is more complex. In exchange for increased complexity, the new program is more general and offers a number of advantages over the original:

- Other tire and engine subclasses can be added which inherit state and behavior from the original tire and engine classes.
- Switching the engine or tires for a Prelude requires modification of only one factory method in **PreludeFactory.**
- Other factories can be added to create other cars.
- Once another concrete factory has been implemented, it is easy to reuse the program to drive this new car.

The following sections illustrates the benefits achieved.

5.3.1 Adding other tire and engine classes

Under the transformed design, new tire and engine classes inherit state and behavior from ***Tire*** and ***Engine*** respectively (Figure 5.10). This inherited state and behavior originally belonged to the **GY184_HR14** and **VTEC2_2** classes. This example adds the **Bridge184_SR14** tire and **AccordEngine** engine classes.

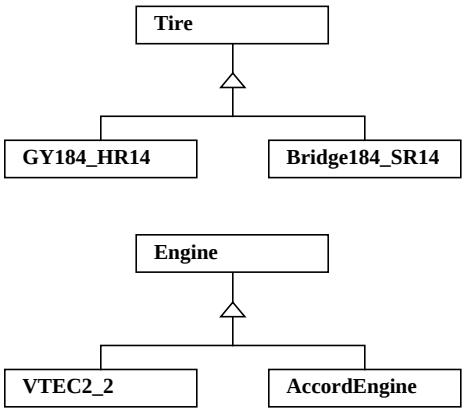


Figure 5.10: Abstract factory created

```
// adding engine used in Honda Accord which inherits from new
// Engine superclass
class AccordEngine : public Engine {
  AccordEngine {
    max_miles = 120000;
    miles = 0;
  }
}

// adding tire used in Honda Accord which inherits from new Tire super-
class.
class Bridge184_SR14 : public Tire {
  Bridge184SR_13 {
    max_miles = 30000;
    miles = 0;
  }
}
```

5.3.2 Switching classes

Switching the tires for a Prelude requires that only one factory method be modified in **PreludeFactory**. The original program would have required more than one dozen changes. The code change below switches the Prelude tire from **GY184_HR14** to **Bridge184_SR14** (Figure 5.11).

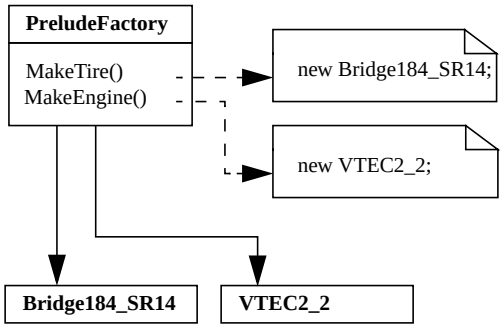


Figure 5.11: PreludeFactory altered to produce Bridge184_SR14 tires

```
class PreludeFactory : public CarFactory {
  Tire *MakeTire() {
   // tire produced by factory method changed from GY184_HR14 to Bridge184_SR14
    return new Bridge184_SR14;
  }
  ...
}
```

5.3.3 Adding factories

Other concrete factories can be defined to create new cars. This example creates a factory for producing Honda Accords (Figure 5.12).

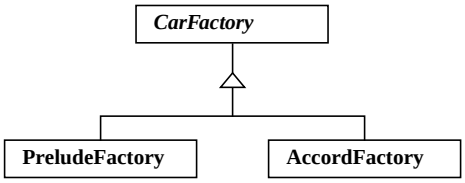


Figure 5.12: Concrete factory added

```
class AccordFactory : public CarFactory {
  Tire *MakeTire() {
    return new Bridge184SR_14;
  }
  Engine *MakeEngine() {
    return new AccordEngine;
  }
}
```

5.3.4 Application reuse

Once other concrete factories have been implemented, it is easy to reuse the application for driving other cars. Modifying the transformed program to create and drive an Accord (instead of a Prelude) involves a change to a single statement in the **App** class initializer (Figure 5.4):

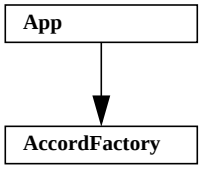


Figure 5.13: App class changed to use AccordFactory

```
car_factory = new AccordFactory;
```

5.4 Summary

This chapter introduces a method for application evolution which first transforms the design while preserving behavior, and then extends the better designed system. Refactorings can automate the design transformation process. Automation significantly reduces, if not eliminates, the burden of identifying and modifying source code to affect design changes. Modifications are done correctly, thereby reducing

costly and tedious debugging and testing that would otherwise have to be performed. In the example presented, refactorings automated all design changes and the newer design is shown to be more extensible and reusable than the original.

The next chapter explores whether the results from this example can scale to tackle the complexities of real-world applications whose code size can exceed 100K lines.

Chapter 6

Evolving real world applications

Our research evaluates whether refactoring technology can successfully transform mainstream applications. In this chapter, we present results on using refactorings to replicate the design evolution of two non-trivial C++ applications. Observations and conclusions are noted and the potential impact of refactorings on design evolution is assessed.

Since refactorings have been shown to support many common forms of design evolution (Chapter 4), the expectation was that many design changes reflected on application class diagrams would be automatable. We did not know if the changes would be localized to a class or block of code in which case they might be easily performed by hand, or if the changes would have a widespread impact on application source code and might be tedious and error-prone if performed manually. Answers can be found by applying refactorings to real C++ applications.

6.1 Unique features

The following features make this study unique:

Replication of design evolution. Designs were extracted from two versions of the same application. The older design became the initial state and the newer

design became the target state. Our objective was to determine if a sequence of refactorings could be applied to transform the initial state to the target state. 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 software systems. Refactoring scalability is tested by transforming large systems. Ideas that appear to be 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 many practical reasons such as backward compatibility with C, portability, availability of third party compilers and tools, legacy system compatibility, and availability of trained personnel. It was expected that C++'s complexity might introduce problems which would not appear for less popular object-oriented languages. A side benefit of this choice is that most claims for C++ can also be made for the increasingly popular Java programming language.

6.2 Application selection

Applications were selected based on the following criteria:

- Access to source code so that designs could be extracted and code could be transformed.
- Application size measured in thousands of lines. The first system studied is relatively small at only 3K lines of source code, however, it undergoes a major design change as a class hierarchy is split in two. The second system is large by most standards with approximately 500K lines of code.

- Availability of multiple versions of the application so that designs between old and new versions could be compared.
- Expectation that some design changes between versions were automatable. The three major types of change we searched for were schema evolutions, design patterns, and hot-spot meta patterns.

The final criteria, expectation that some design changes were automatable, deserves further discussion. A number of development activities can occur between versions which may not impact an application's class diagram: other aspects of the design may change (e.g. state diagrams, object interaction diagrams, and scenarios), algorithms may be changed, functionality can be added within the framework of the existing design, etc. One software system we considered met the first three criteria but not the last.[1]

6.3 Evolving CIM Works

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

CIM Works provides a graphical user interface for creating, connecting, and testing objects used in the production of semiconductors. Major design changes in CIM Works occur between Version 2 and Version 4. The Version 2 design shown in Figure 6.1 stores data and its graphical representation in the same object. For

1. This was the Interviews user interface toolkit [Lin92] whose design did not change significantly between the two versions studied (Version 3.0 and Version 3.1). Note that toolkit designs need to be relatively stable between versions so that applications using the toolkit will be minimally impacted when upgrades are made.

Figure 6.1: Version 2 Architecture

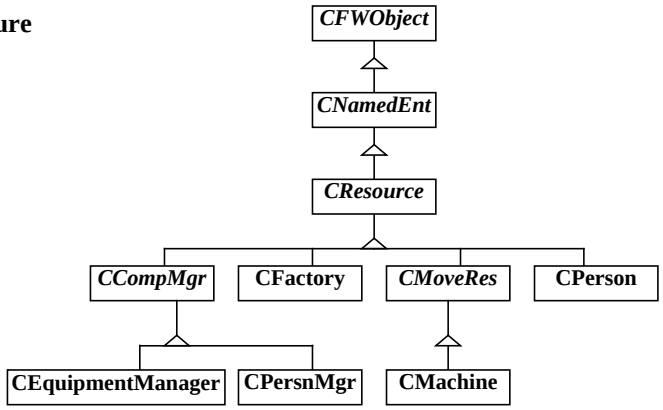
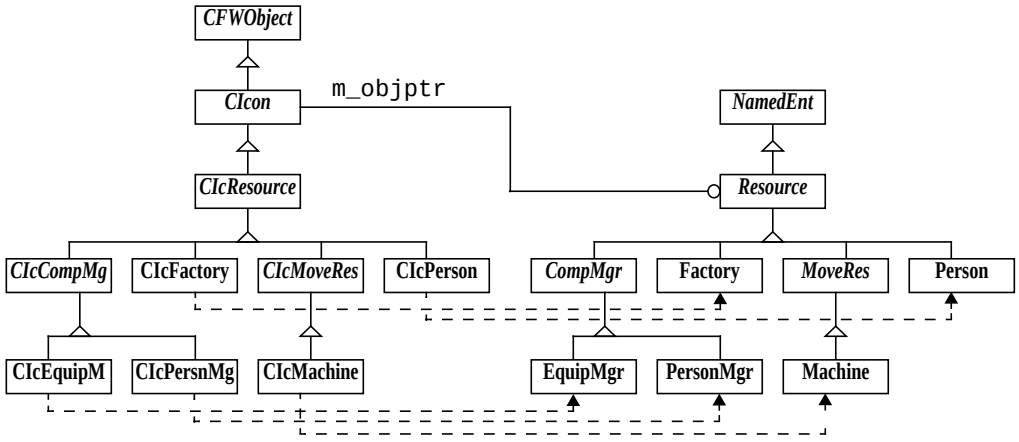


Figure 6.2: Version 4 Architecture



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 6.2 separates data and graphics into two class hierarchies. For example, class **CEquipmentManager** was split into classes **CIcEquipmentManager** and **EquipmentManager**. This separation gave Version 4 the freedom to create different views of the same data as with the model-view-controller paradigm [Kra88]. Version 2 was approximately 3K lines of source code. The actual size of the program transformed (including Microsoft header

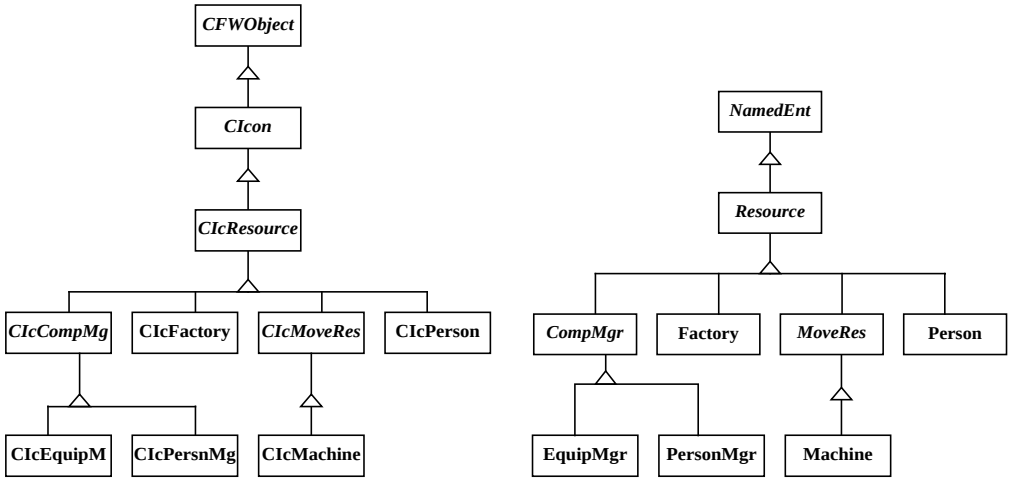


Figure 6.3: Original classes renamed and Data Classes created

files and preprocessing) was 11K lines of code.

6.3.1 Refactoring steps

The transformation between designs is accomplished in seven steps, each of which is realized by applying a sequence of primitive refactorings:

1. *Rename the class hierarchy to match the GUI classes in the new design.* Classes are renamed using **rename_class**. The original classes retain the GUI aspects of objects, whereas their corresponding "split" classes — created in Step 2 through 5 — encapsulates object data.
2. *Create the new data class hierarchy.* **Factory**, **Person**, **EquipmentManager**, etc. are created using **create_class**. Inheritance relationships between the classes are added with **inherit** (Figure 6.3). The newly created data classes will contain the data portion of the split hierarchy. Note that at this point, the classes are empty.
3. *Link each concrete GUI class to its corresponding data class.* `m_objptr`

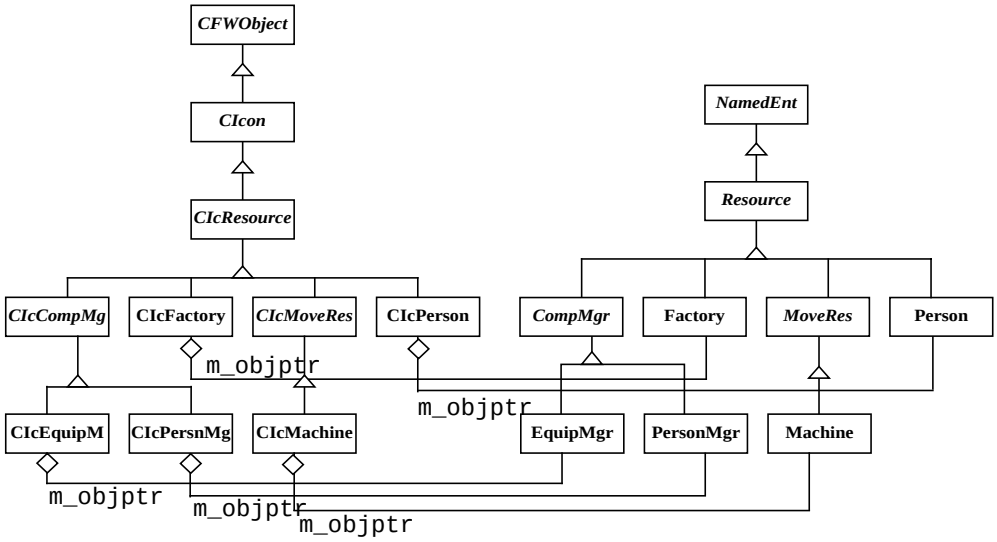


Figure 6.4: Connect GUI and Data classes

instance variables are added to the concrete GUI classes using **add_variable**. `m_objptr` is of the corresponding data class type (Figure 6.4).

4. *Move data instance variables and methods from concrete GUI classes to their corresponding data classes.* Variables and methods are moved from GUI classes to data classes using **move_variable_across_object_boundary** and **move_method_across_object_boundary**. For example, the instance variables `shift` and `dept` and the methods `getShift()` and `getDept()` in **CIcPerson** are moved to **Person** (Figure 6.5). Data is accessed through the `m_objptr` instance variable. For the `shift` instance variable, a reference to `(CIcPerson *) person_ptr->shift` is transformed to `(CIcPerson *) person_ptr->m_objptr->shift`.
5. *Move common instance variables and method declarations up the data class hierarchy.* Methods and variables are moved using **pull_up_variable** and **declare_virtual_method** (Figure 6.6).

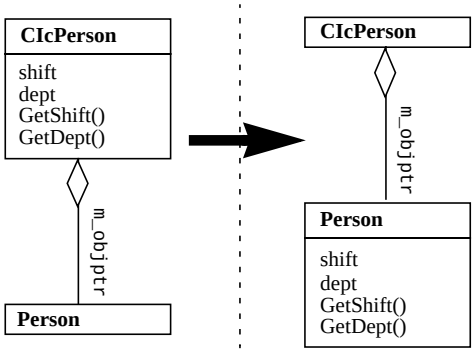


Figure 6.5: Instance variables and methods moved to data classes

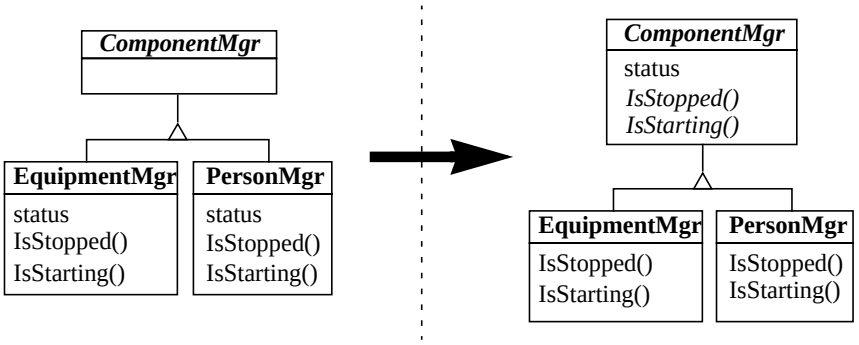


Figure 6.6: Instance variables and method declarations moved to abstract classes

6. *Change m_objptr from a structure to a pointer* (Figure 6.7). `m_objptr` is initially created as a structure to guarantee a 1:1 correspondence between a GUI object and a data object. This allows instance variables and methods of a GUI class to be safely moved to their corresponding data class. **structure_to_pointer** converts `m_objptr` from a structure to a data object pointer which is initialized in the GUI constructor.
7. *Declare the reference between GUI objects and data objects in the abstract classes.* References to data objects are made abstract[2] (Figure 6.2). This completes the design transformation.

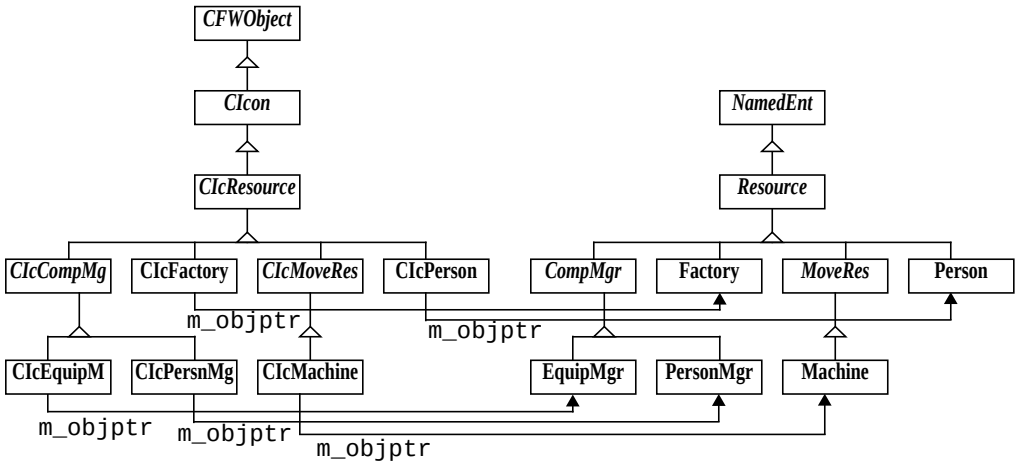


Figure 6.7: m_objptr changed from a structure to a pointer

6.3.2 Lessons learned

The design evolution of CIM Works was accomplished using 81 individual refactorings resulting in a total of 486 lines of source being modified. The transformed software was tested and performed identically to the original software. In the process of using refactorings to duplicate the design evolution of CIM Works, we made the following observations and conclusions:

Automating design changes. The most important result of this experiment is to establish that refactorings can automate significant design changes in a real world application. Between Version 2 and Version 4, the CIM Works class

---

2. 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->m_objptr->f_name = "John";
```

is transformed to:

```
CIcPerson *p = new CIcPerson;
((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.

hierarchy was split into two connected hierarchies requiring code changes spread throughout the code base. All changes were automatable with refactorings, although as noted, no refactoring was created to support step 7.

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 majority of the refactoring time savings would result from automating the location of affected code and reducing the testing time for the new design.

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 this example, the number of refactorings was large although the conceptual number of transformation steps was small. One way to reduce the number of refactorings would be to provide larger grain refactorings. In this example, the number of refactorings would be significantly reduced if refactorings to move multiple variables and methods were available (see discussion on Granularity of Transformations in Section 9.2).

Preprocessor directives. One of our early realizations was that a C++ program transformation tool cannot deal with preprocessor directives because preprocessor directives are not part of the C++ language. CIM Works uses Microsoft Windows constants which must be preprocessed to produce a compilable program. Partial solutions which would solve the problems for this example are proposed in Section 8.2, however, we do not see a general way to handle all occurrences of preprocessor information embedded in C++ code.

Computer formatting. Under our program transformation model, source code is parsed into an intermediate representation, transformed, and unparsed to

produce a new version of the source code. The original formatting information is not preserved in the intermediate representation, thus, all formatting is computer generated.

Source file access. In transforming this Windows application, it occurred to us that enabling conditions are currently written with the assumption that all source code for a program is transformable. This may not be the case. For example, a user may attempt to move an instance variable from a user-defined subclass to a Microsoft Foundation Class which has a read-only header file and no available source. This was not an issue here because the design changes being replicated with refactorings were limited to developer code. For a transformation system targeting a mainstream development environment however, checks must be made to determine if any files affected by a transformation are read-only.

Generality and scalability. We believe that the results obtained from refactoring CIM Works are not at all unusual. Breaking classes into pieces is common in the evolution of a design as is the creation and population of abstract classes. It is also our belief that if CIM Works were to grow to tens of thousands of lines before applying the same design changes, the lines of code affected would approach linear growth proportional to the size of the application. This assumes that new code added to CIM Works would continue to use the data instance variables and methods from the original class hierarchy and would need to be refactored when the hierarchy was split.

Uniqueness of derivation. Some transformations are ordered — instance variables and methods must be moved to the new data classes before they can be moved up the inheritance hierarchy. In general, however, the derivation of a new design is not unique. For CIM Works, instance variables and methods could have been moved to the new data classes before their superclasses were created. It may be appropriate to provide large grain transformations to replace a recurring series

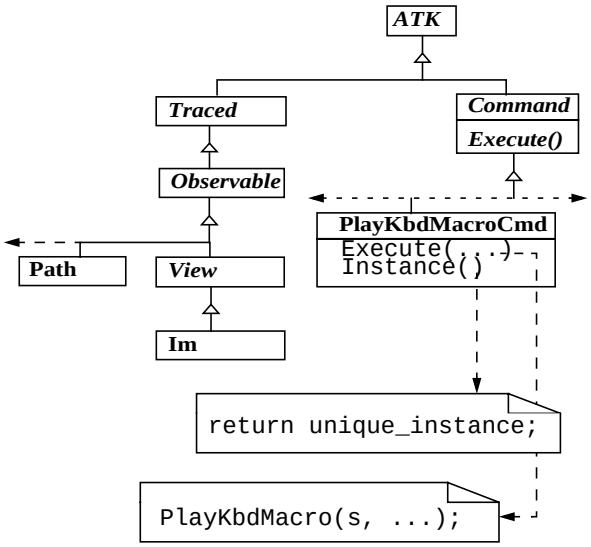


Figure 6.8: Software microarchitecture for AUIS Version 8.0

of primitive transformations (such as those that might implement a design pattern). This would simplify the refactoring selection process and reduce the total number of refactorings required to execute a design change.

6.4 Evolving the Andrew User Interface System

The *Andrew User Interface System (AUIS)* from CMU is an integrated set of tools that allows users to create, use, and mail documents. Documents are constructed using an extensible compound document architecture. This architecture combines everything from text to pictures and graphs to spreadsheets to figures into a single document [Mor85].

The two versions under study were Version 6.3 written in C and Version 8.0 converted to C++. Version 6.3 stores actions on AUIS objects as function pointers. No class diagram is shown for this version because it is written in C and has no classes or inheritance. Version 8.0 shown in Figure 6.8 supports and

recommends creation of a separate subclass for each action (similar to the Command design pattern[3] [Gam95]).

6.4.1 Refactoring steps

AUIS is approximately 500K lines of code. The change from function pointers to commands affects ninety classes using almost 800 actions. The transformation is accomplished in four steps, each of which is realized by applying a sequence of primitive refactorings:

1. *Convert Version 6.3 C structures to C++ classes.* Structures are converted to classes using **structure_to_class**. We can now display a class diagram for the application Figure 6.9.[4]

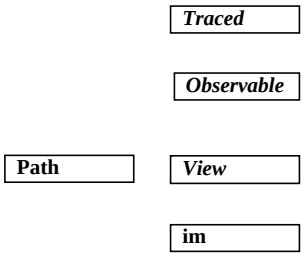


Figure 6.9: Structures converted to classes

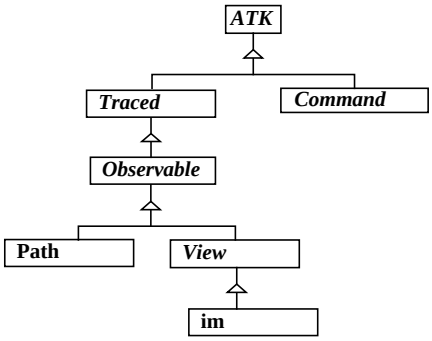


Figure 6.10: Class hierarchy created

2. *Add abstract classes to the class hierarchy.* Classes **ATK** and **Command** are created using **create_class**. Inheritance relationships between classes are added using **inherit** (Figure 6.10).

---

3. The Command design pattern objectifies an action. The action is triggered by calling an `Execute()` method implemented in each derived class [Gam95].
4. Version 6.3 is written in C with preprocessor extensions to support classes. Refactorings which convert structures to classes and functions to methods were not applied whenever possible because these transformations are atypical and would affect almost every function call in the program. Only structures needed to support the Command pattern were transformed.

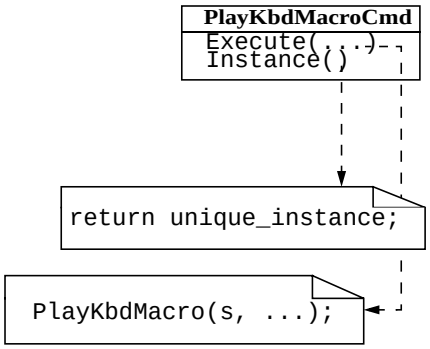


Figure 6.11: PlayKbdMacro converted to a command class

3. *Derive command classes for each action.* Command classes are created from each action procedure using **procedure_to_command**. Figure 6.11 displays the result of transforming the procedure `PlayKbdMacro()` into a command class. The newly created **PlayKbdMacroCmd** class contains an `Execute()` method which calls `PlayKbdMacro()`. 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.
4. *Convert procedure pointers to commands.* Procedure pointers are converted 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 the `Instance()` method. Figure 6.12 displays the transformation of the **bind_Description** structure. The `proc` instance variable is converted to a ***Command*** pointer, `(*e->proc)(...)` is converted to `e->proc->Execute(...)`, and the function pointer `PlayKeyboardMacro` is converted to `PlayKeyboardMacroCmd::Instance()`.[5]

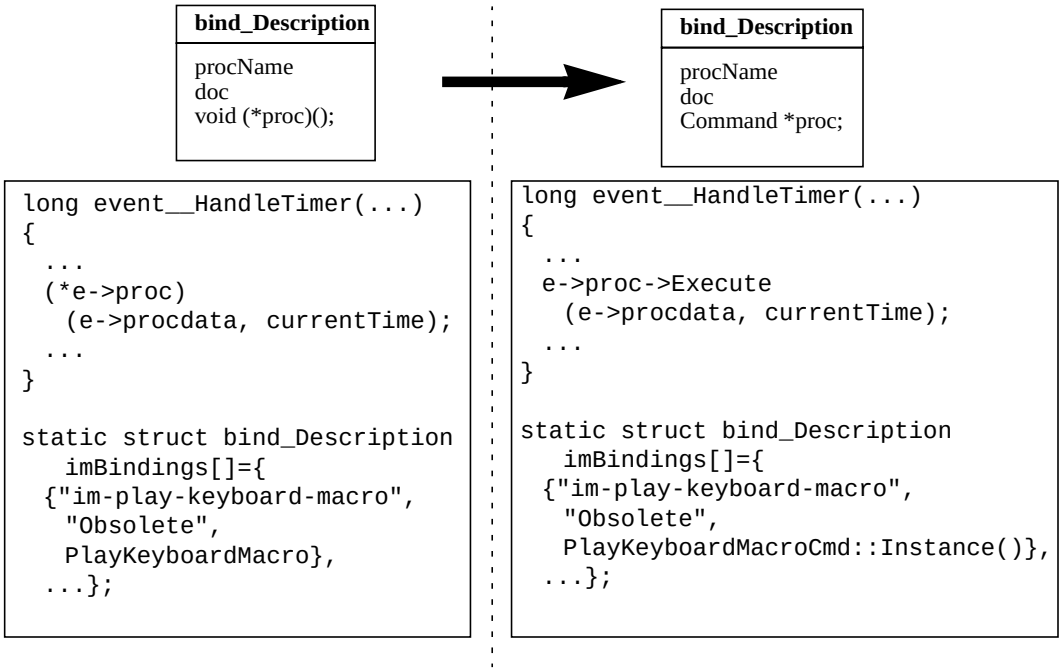


Figure 6.12: Convert procedure pointer to Command pointer

6.4.2 Lessons learned

The design evolution of AUIS was accomplished using 800 refactorings resulting in 14K lines of code changes. In the process of using refactorings to duplicate the design evolution of AUIS, we made the following observations and conclusions:

Automating design changes. The most important result of this experiment is to establish that refactoring technology can scale to applications greater than 100K lines of code. We believe that AUIS is the largest application to be refactored to date. All changes were automatable.

5. The Version 8.0 implementation differs from the result obtained with refactorings. In Version 8.0, reference counts were used to track the number of pointers to each action object and it was possible to have multiple objects representing the same action. Because the action objects were replacing function pointers with unique identities, we imposed the Singleton design pattern on each action to guarantee that there would be at most one copy of any action object.

We estimate that the AUIS changes would take us two weeks to implement and debug by hand versus one day when aided by refactorings. In the CIM Works experiment the majority of the refactoring time savings would result from automating the location of affected code and reducing the testing time for the new design. For the AUIS experiment, we believe that the majority of the refactoring time savings would result from the automated execution of changes since there are over 800 actions and the conversion of a single action to a command class requires the addition or modification of eighteen lines of code.

Experimentation. More than one implementation of command classes is possible. With refactorings, it is possible to change the choice of refactorings or refactoring arguments to produce different implementations. With manual coding, it is very difficult and tedious to make even the smallest change to a command class implementation since there are over 800 actions spread through ninety classes.

Granularity of transformations. As with CIM Works, we did not attempt to minimize the number of refactorings required. Despite the large number of refactorings and the large number of lines of code affected, the design changes to AUIS are conceptually simpler (only four steps) than the design changes to Car Factory and CIM Works. Larger grain refactorings could significantly reduce the number of refactorings required (see discussion on Granularity of Transformations in Section 9.2). In this example, most of the refactorings take place in Step 3 — converting action procedures to commands. A larger grain refactorings which converted a list of procedures to commands could execute Step 3 in a single transformation. This would reduce the total number of refactorings to fewer than twenty.

Program families. Parnas argued that software developers should design each program as a member of a family of programs [Par79]. AUIS is a toolkit delivered

with multiple target applications. We recognized that when you transform a file used by more than one program, it would be desirable for the transformations system to check enabling conditions for all programs which use that file. Otherwise, a file might be transformed safely for one program while causing another program which uses the same file to break.

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

Conditional compilation. The CIM Works example introduced the C++ preprocessor problem. The AUIS example complicates this problem with function-like macros and conditionals compilation flags:

`#define macro(x) F(x)` — function-like macros can sometimes be replaced by an inline function whose return type must be known. However, for function-like macros which used the `#` and `##` operators, it was not be possible to create an equivalent inline function and the call sites to the macro were changed.

`#define FLAG` and `#ifdef` — conditionally compiled information is difficult to maintain. Our implementation chooses a single set of compiler flags in the preprocessing stage. Conditionally compiled code which is removed by the flags is thus permanently lost. Preserving this code is a difficult problem discussed further in Section 8.2.

Preserving comments. A requirement for any source to source transformation system is that source code comments must be preserved. In the Andrew example, comments accounted for more than 20% of all non-blank lines. The Sage++

toolkit used in this research demonstrates that it is possible to preserve comments while refactoring C++ programs [Bod94]. Comments can also be automatically inserted to document changes resulting from refactorings.

Opdyke's invariants for behavior preservation. A significant discovery while refactoring AUIS is that the refactorings from Opdyke do not preserve behavior in all cases. Opdyke identifies seven invariants to be preserved by each transformation, however, there is no guarantee that preserving these invariants preserves behavior. The following is a counterexample based on an actual failure when attempting to transform AUIS:

```
//addition of z iv causes core dump
class testclass {
public:
  char * x;
  int y;
//  int z;
};

testclass bs[] = {"spit", 3, "bye", 5};
```

Calling the **add_variable** refactoring to add instance variable `z` to **testclass** causes the above code fragment to core dump because the initializer list assigns the string `"bye"` to the integer `z`.[6] Exceptions are more likely to appear in a language such as C++ with its complex syntax and in large software systems such as AUIS. This discovery has far reaching implications from the definition of a refactoring as a behavior-preserving transformation (Do refactorings even exist

6. Opdyke states that the footnote in Section 5.1 of [Opdyke 92] should be generalized from "programs that *test* the physical size of objects could see their behavior change" to "programs that *depend upon* the physical layout of objects could see their behavior change" to correct this problem. Initializer lists were not addressed in Opdyke's work. In general, a refactoring from [Opdyke 92] which transforms a class from an aggregate to a non-aggregate will not be behavior preserving for programs which use initializer lists because C++ does not allow you to use initializer lists on non-aggregates [Ellis 90].

   The main point remains that there is no mathematical guarantee that a refactoring preserving Opdyke's invariants will preserve behavior.

since they are defined to be behavior-preserving but there is no proof of this property?) to questions about the claims of reduced testing requirements (Don't you still have to test since you can't guarantee that behavior is preserved?) Our views on the issue of behavior-preservation are discussed in Section 8.2 and related work on this topic is presented in Section 2.5.

Generality and scalability. We believe that the results obtained from refactoring AUIS are typical of applying a design pattern solution regarding the number of classes affected (90) although few patterns would affect as many functions (800). The number of lines of code affected by the type change in **bind_Description** may be atypically small because this structure was only set and accessed in two places whereas it is possible for the number of accesses to grow linearly with the size of the program.

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

6.5 Summary

Given that refactorings can automate many common forms of design evolution, the expectation was that many design changes experienced by the applications under study would be automatable. In fact, all changes except for Step 9 of the CIM Works example were automated and as noted on this step, automation appeared

possible (although not necessarily desirable). These results are encouraging because there is an obvious benefit if refactorings can automate a majority of all design changes.

Refactorings automated changes which were not localized to a single class or block of code. In the CIM Works example, all lines of code which accessed instance variables moved to the data hierarchy were changed. In the AUIS example, the conversion of function pointers to commands required thousands of lines of changes spread throughout ninety classes in the source hierarchy. Refactorings are more likely to be used if they perform non-localized changes because non-localized changes tend to be more time-consuming and error-prone.

On the issue of practicality, we found refactorings for C++ to be difficult if not impractical to implement because of the presence of preprocessor information commonly found in most real world applications. Other key issues related to technology transfer include transforming program families, source file access, formatting, and comment preservation. Finally, it was our experiment on a half-million line C++ software system which uncovered a flaw in Opdyke's model for behavior preservation.

Chapter 7

Implementation details

Our experiments transformed legal C++ programs to equivalent C++ programs with an evolved design. This chapter discusses the design considerations for our refactoring implementation, the implementation of individual refactorings, the transformation of applications through a series of refactorings, and an evaluation of the tools we used.

7.1 Design considerations

The research was focused on mainstream development environments. The most popular C++ development environments are Microsoft Visual C++™ for the PC and vi/emacs with sccs (source code control system) for Unix. Rational Rose™ and OMT Professional Workbench™ are the most popular tools for object modeling although they are not currently considered to be essential for an object-oriented development environment. Given the desire to support both PC and Unix development (CIM Works is a PC application and AUIS runs on Unix), we chose to implement refactorings as command line executables. The only requirement placed on an environment is that is must provide direct access to source code. The over-

whelming majority of all C++ development environments provide this access. It was our intent to use a publicly available C++ parser/code generator to aid in the refactoring implementation. We considered four tools:

- Indiana University's Sage++ toolkit [Bod94]. Sage++ provided the advantages of a semantic analyzer and an object-oriented programmer's interface for modifying programs.
- Edison Design Group's C++ Front End[1]. Edison offered a commercial quality implementation but no semantic analyzer.
- Microsoft Research's Intensional Programming tool (IP) [Sim98]. IP provides a powerful environment for implementing program transformations, however, at the time of our selection, only the C language was supported and support for C++ was not planned for another six months[2].
- Brown University's CPPP [Rei94]. CPPP offered a semantic analyzer but no code generator.

The Sage++ toolkit was chosen for its semantic analysis and object-oriented programmer's interface. It was used to successfully transform the Car Factory example presented in Section 5.2. Further work with this toolkit, however, revealed that it was not robust enough to support the transformation of real-world programs. We noted two major deficiencies:

- The Sage++ process of parsing immediately followed by code generation may introduce errors for some legal C++ programs. The expectation is that parsing followed by code generation should have no effect.
- Parsing and analyzing a large program can cause Sage++ to hang or crash. This discovery was made when attempting to transform the 500K AUIS example.

1. For further information on this tool, see the Edison Design Group web site http://www.edg.com.
2. Since that time, support for C++ was removed as a priority goal for the IP project.

Based on these deficiencies and the preprocessor directive problem identified in Section 6.3.2, it was determined that refactorings would be developed only to further the research goal of replicating the design evolution performed by humans and not for the purpose of general public release.

With no plans for a public release, we opted to use the compiler when possible to verify that refactoring invariants had been preserved. Six of Opdyke's seven invariants can be verified by a compiler [Opd92]. In a commercial refactoring implementation, recompiling an application after each refactoring would be too time consuming to be practical. For our purpose of determining whether refactorings can perform design changes previously executed by humans, recompilation after each refactoring was tolerable and saved significant development effort[3].

7.2 Evolving applications

To evolve an application, the original source files are first manipulated to produce a single application file app.C and a .dep file app.dep. The *.dep* representation is a proprietary format used internally by the Sage++ toolkit which stores the parse tree and semantic analysis of the original program. Sage++ provides utilities to convert a C++ program to a .dep file. A series of refactorings placed in a refactoring script is then applied to app.C and app.dep. Figure 6.1 depicts the complete refactoring process:

- The application is prepared for transformation (see details in Section 7.2.1). The output of the process is a C++ application file with its corresponding .dep file.

3. In practice, we used the Sage++ conversion to the .dep format to perform compiler checks. This operation is faster than compiling but still requires that all source code be parsed and analyzed at each step.

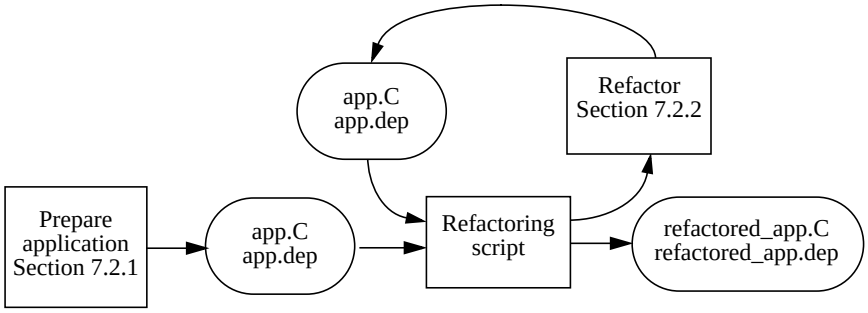


Figure 6.1: Evolving an application

- The refactoring script takes a C++ application file with its corresponding .dep file as input and outputs a refactored C++ application file with its corresponding .dep file. The script transforms an application by calling a series of refactorings (see details in Section 7.2.2).

The refactoring script checkpointed the application after each refactoring to allow for comparison with the original program. Checkpointing also made it possible to recover from an enabling condition violation by changing the script and continuing from the last successfully applied refactoring. The script used to evolve CIM Works is displayed in Figure 6.2a and Figure 6.2b.

7.2.1 Preparing the application

In their original form, the applications we evolved were distributed among many source files: they contained preprocessor information, they were formatted by humans, and in many cases they could not be read properly due to limitations of the Sage++ toolkit. Applications were prepared for transformation with the following steps (Figure 6.3):

1. Pack source files into a single file — app_a.C. Ideally, a refactoring should take a makefile target as an argument. For simplicity, we eliminated the need for a makefile by packing all source files into a single

```
refactor rename CNamedEntity CIcon
refactor rename CResource CIcResource
refactor rename CComponentManager CIcComponentManager
refactor rename CEquipmentManager CIcEquipmentManager
refactor rename CPersonManager CIcPersonManager
refactor rename CFactory CIcFactory
refactor rename CMovementResource CIcMovementResource
refactor rename CMachine CIcMachine
refactor rename CPerson CIcPerson

refactor movedown CIcResource M nameQualifiedTo M
       showNameQualifiedTo V owner V qualLevel
refactor movedown CIcComponentManager M nameQualifiedTo M
       showNameQualifiedTo V owner V qualLevel
refactor movedown CIcMovementResource M nameQualifiedTo M
       showNameQualifiedTo V owner V qualLevel

refactor add_class NamedEntity
refactor add_class Resource
refactor inherit NamedEntity Resource
refactor add_class ComponentManager
refactor inherit Resource ComponentManager
refactor add_class EquipmentManager
refactor inherit ComponentManager EquipmentManager
refactor add_class PersonManager
refactor inherit ComponentManager PersonManager
refactor add_class Factory
refactor inherit Resource Factory
refactor add_class MovementResource
refactor inherit Resource MovementResource
refactor add_class Machine
refactor inherit MovementResource Machine
refactor add_class Person
refactor inherit Resource Person

refactor createiv CIcEquipmentManager m_objptr EquipmentManager
refactor createiv CIcPersonManager m_objptr PersonManager
refactor createiv CIcMachine m_objptr Machine
refactor createiv CIcFactory m_objptr Factory
refactor createiv CIcPerson m_objptr Person

refactor moveclasspos EquipmentManager CIcEquipmentManager
refactor moveclasspos PersonManager CIcPersonManager
refactor moveclasspos Machine CIcMachine
refactor moveclasspos Factory CIcFactory
refactor moveclasspos Person CIcPerson

refactor moveivtoclass CIcEquipmentManager owner m_objptr
refactor moveivtoclass CIcEquipmentManager qualLevel m_objptr
refactor movemethod CIcEquipmentManager nameQualifiedTo m_objptr
refactor movemethod CIcEquipmentManager showNameQualifiedTo m_objptr
refactor moveivtoclass CIcEquipmentManager machineList m_objptr
refactor movemethod CIcEquipmentManager addMachine m_objptr
refactor movemethod CIcEquipmentManager removeMachine m_objptr
refactor movemethod CIcEquipmentManager allMach m_objptr
```

Figure 6.2a: Script to transform CIM Works

```
refactor moveivtoclass CIcPersonManager owner m_objptr
refactor moveivtoclass CIcPersonManager qualLevel m_objptr
refactor movemethod CIcPersonManager nameQualifiedTo m_objptr
refactor movemethod CIcPersonManager showNameQualifiedTo m_objptr

refactor moveivtoclass CIcMachine owner m_objptr
refactor moveivtoclass CIcMachine qualLevel m_objptr
refactor movemethod CIcMachine nameQualifiedTo m_objptr
refactor movemethod CIcMachine showNameQualifiedTo m_objptr
refactor moveivtoclass CIcMachine description m_objptr
refactor moveivtoclass CIcMachine vendor m_objptr
refactor moveivtoclass CIcMachine modelNumber m_objptr
refactor moveivtoclass CIcMachine serialNumber m_objptr
refactor moveivtoclass CIcMachine softwareVersionNumber m_objptr
refactor movemethod CIcMachine getSoftwareVersionNumber m_objptr
refactor movemethod CIcMachine getSerialNumber m_objptr
refactor movemethod CIcMachine getModelNumber m_objptr
refactor movemethod CIcMachine getVendor m_objptr
refactor movemethod CIcMachine getDescription m_objptr
refactor movemethod CIcMachine setSoftwareVersionNumber m_objptr
refactor movemethod CIcMachine setSerialNumber m_objptr
refactor movemethod CIcMachine setModelNumber m_objptr
refactor movemethod CIcMachine setVendor m_objptr
refactor movemethod CIcMachine setDescription m_objptr
refactor movemethod CIcMachine isMaterialTransporter m_objptr
refactor movemethod CIcMachine isProcessingMachine m_objptr
refactor movemethod CIcMachine isStorageUnit m_objptr

refactor moveivtoclass CIcFactory owner m_objptr
refactor moveivtoclass CIcFactory qualLevel m_objptr
refactor movemethod CIcFactory nameQualifiedTo m_objptr
refactor movemethod CIcFactory showNameQualifiedTo m_objptr
refactor moveivtoclass CIcFactory equipmentManager m_objptr
refactor moveivtoclass CIcFactory personManager m_objptr
refactor movemethod CIcFactory startup m_objptr
refactor movemethod CIcFactory goToStandby m_objptr
refactor movemethod CIcFactory shutdownImmediate m_objptr
refactor movemethod CIcFactory shutdownNormal m_objptr
refactor movemethod CIcFactory registerManager m_objptr
refactor movemethod CIcFactory removeRegistrationForManager m_objptr
refactor movemethod CIcFactory componentStartupComplete m_objptr
refactor movemethod CIcFactory componentShutdownComplete m_objptr
refactor movemethod CIcFactory allMachines m_objptr
refactor movemethod CIcFactory allStorageUnits m_objptr
refactor movemethod CIcFactory allMaterialTransporters m_objptr
refactor movemethod CIcFactory getPersonManager m_objptr
refactor movemethod CIcFactory getEquipmentManager m_objptr
```

Figure 6.2b: Script to transform CIM Works

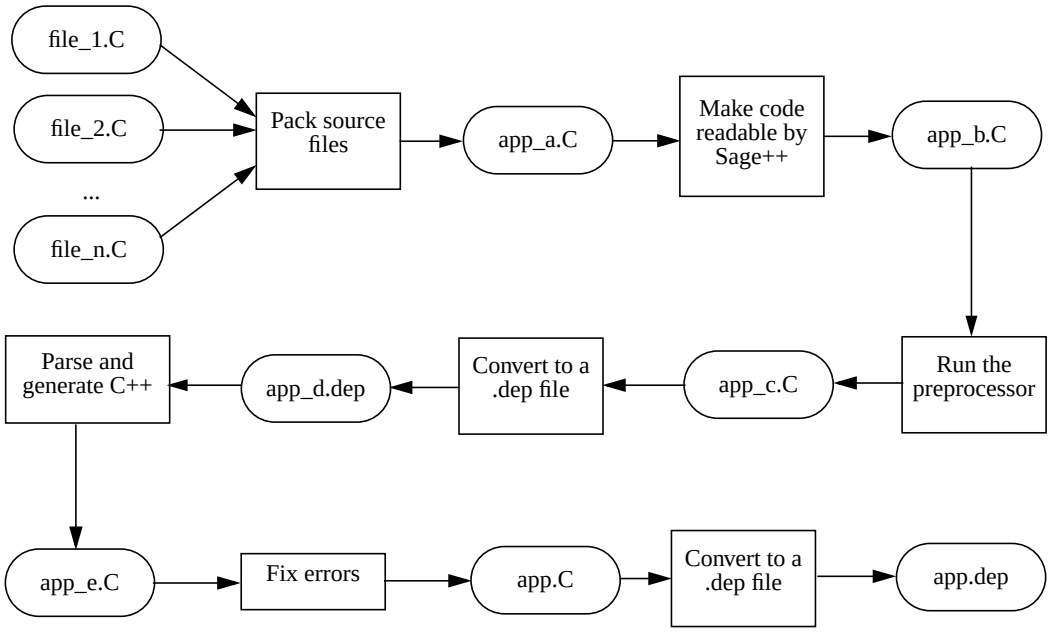


Figure 6.3: Generating the program to be refactored

application file. This issue is discussed further in Section 8.3.

2. Make any changes required for Sage++ to read app_a.C and save the modified file in app_b.C. For example, Sage++ does not support initilization of class variables as presented in Ellis and Stroustup [Ell90 page 150]. Initializations of this kind had to be rewritten. app_b.C represents the base program being transformed.
3. Run app_b.C through the preprocessor to produce app_c.C. This version of the program is expressed entirely in C++ and contains no preprocessor information.
4. Convert app_c.C to the Sage++ .dep format to produce app_d.dep. This is a version of the program readable by the Sage++ toolkit.
5. Use Sage++ to read app_d.dep and generate a C++ file app_e.C. This version of the program is *theoretically* equivalent to app_c.C from Step 3 except for its computer formatting. In practice, the Sage++ code

generation process may introduce errors.

6. As a postprocess, run a program specific clean-up script called fix_errors.exe on app_e.C to produce a legal program app.C. This version of the program is equivalent to app_c.C from Step 3 except for its computer formatting.
7. Create a .dep version of app.C using Sage++ utilities.

The resulting app.C possesses three properties:

- It is equivalent to app_b.C, the base version created in Step 2.
- It can be successfully converted to a .dep file using Sage++ utilities.
- It is formatted using the Sage++ code generation routines so that the Unix "diff" utility can be used to compare this base version and other refactored versions of the program. The changes identified by diff were used to assess the impact of design changes on refactored code.

7.2.2 Implementation of refactorings

The input to a refactoring was defined to be a C++ program with its corresponding .dep representation. The output of a refactoring was defined to be a semantically equivalent transformed C++ program with its corresponding .dep representation.

When a refactoring succeeds, a program is transformed as prescribed by the refactoring. When a refactoring fails, it implies that some refactoring enabling condition has been violated. The steps to perform an individual refactoring are shown in Figure 6.4:

1. Verify all enabling conditions which are not checked automatically. If enabling conditions are not met, then the refactoring fails. Enabling conditions which were checked by hand in our refactoring implementation are noted for each refactoring in Appendix A.
2. Run a routine written for each refactoring. The routine takes a .dep file and

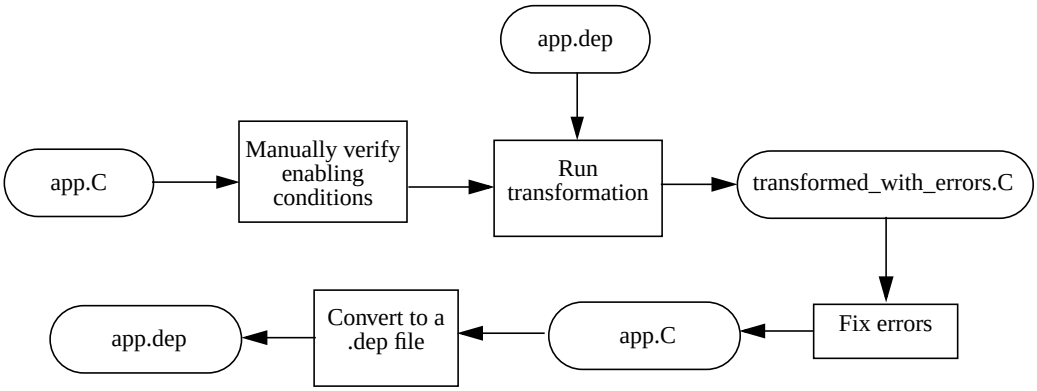


Figure 6.4: Refactoring steps

refactoring arguments as input, and outputs a transformed C++ program. If this routine fails, then the refactoring fails.

3. Run fix_errors.exe to fix any errors introduced by Sage++'s parsing and code generation process to produce a legal refactored C++ program. Ideally, the program output from the previous step should be legal, however, it is possible for Sage++ to introduce program errors which need to be corrected. An example is that under some conditions, a single occurrence of the "`typedef`" keyword in the original program would appear twice as "`typedef typedef`" in the generated code leading to a compiler error[4].
4. Convert the repaired program to a .dep file with Sage++ utilities. In the process of creating a .dep file, Sage++ performs a semantic analysis of the program. This step replaces a compiler pass for the purpose of determining if invariants have been violated (as discussed in Section 7.1 — Design Considerations). If a .dep file cannot be produced, then the refactoring fails. Note that when a refactoring has been successfully completed, both

4. The two other leading sources of error were the failure to preserve enum typedefs and the addition of extra levels of parenthesis to certain expressions (e.g. converting "cmd" to "(cmd)").

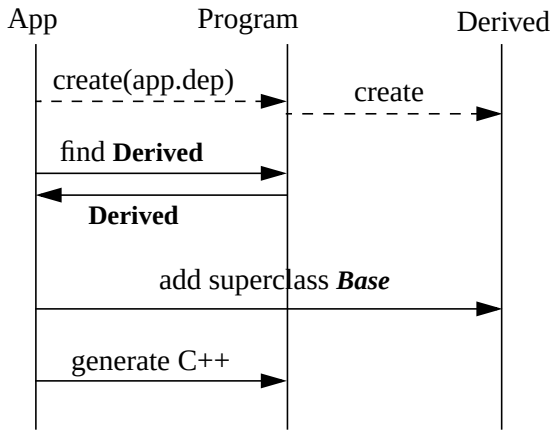


Figure 6.5: Event trace for inherit

the transformed C++ program and its .dep are representation are generated. These outputs match the refactoring input requirements when performing a series of refactorings.

The routine for each refactoring from Step 2 is written in C++ using Sage++'s programmer interface to read, transform, and generate code. As an example, the event trace for the **inherit** refactoring is displayed in Figure 6.5. The vertical lines of the event trace represent objects which participate in the **inherit** refactoring. Lines between objects represent method calls.

1. The main **App** object takes a .dep file as input and creates a **Program** object. When **Program** is initialized, it reads and initializes all class objects in the .dep file including **Derived**.
2. **App** searches the **Program** for the **Derived** class.
3. **Base** is added as a superclass of **Derived** using the `add_superclass` method.
4. C++ is generated for the transformed program.

This trace illustrates the benefit of the Sage++ toolkit. The input .dep file is converted to a program object which contains C++ objects including class objects. The three other transformation tools considered (C++ Front End, IP, and CPPP)

all require the transformation developer to perform surgery on a tree representation of a program being transformed. To use these tools, the refactoring writer must know the details of how inheritance is represented in a tree. Sage++ provides an `add_superclass` method for class objects. Thus, a transformation developer can add a superclass without knowledge of how Sage++ represents inheritance internally.

7.3 Sage++ for transformation developers

Our experience with Sage++ provides some insight on tools which support source code transformations. We offer the following observations on Sage++ in particular and on transformation support tools in general:

- A commercial quality refactoring tool requires a robust parser implementation. Sage++ is a university project which is clearly not a starting point for a commercial tool. A list of known bugs can be found at the Sage++ web site `http://www.extreme.indiana.edu/sage/sage-bugs/maillist.html`.
- A commercial quality refactoring tool requires a full implementation of the language specification. The Sage++ toolkit attempted to hide the abstract syntax tree but did not provide, for example, an interface for declaring a pure virtual function or support for some forms of class variable initialization. One version of the toolkit did not provide a way to create a public function.
- Sage++ does not provide a pattern matching capability to aid in locating code to be transformed. An example where this capability would have been useful is in the **structure_to_pointer** refactoring which required that code of the form `struct.variable` be identified. Sage++ can have up to five differ-

ent representations of this pattern embedded in the abstract syntax tree. To identify affected code, the internal Sage++ tree representation for all five forms must be known and searched for using low-level primitives.

- Generation of new code was generally very tedious. Sage++ requires the transformation developer to define or search for individual elements of a program statement and to compose them together. A more convenient method for generating code is demonstrated by the Jakarta Tool Suite [Bat98] which supports insertion of program fragments written in the native language.

Chapter 8

Introspection and lessons learned

This chapter presents refactoring benefits, limitations, and requirements.

8.1 Refactoring benefits

Automating Modes of Object-Oriented Software Evolution. Opdyke proposed refactorings to automate the OODBMS schema transformations identified by Banerjee and Kim. More recent work in the object-oriented community seeks to identify recurring patterns in the designs of experienced object-oriented designers. Chapter 4 demonstrates that many of Gamma's design patterns and Pree's hot-spot meta patterns can be viewed as program transformations applied to an evolving design. Furthermore, these transformations can be automated with refactorings.

Our research also recognizes patterns which are fundamental to a software design. Fundamental patterns cannot be added with refactorings and should be considered for inclusion in the initial software design to avoid expensive design evolution costs.

Evolving Applications. In Chapters 5 and 6, refactorings were used successfully to evolve three applications. For Car Factory, the Abstract Factory design pattern was added to support multiple car classes and multiple component

classes. For CIM Works, the main class hierarchy was split into two connected hierarchies in an automated way. For AUIS, procedure pointers were converted to use the Command design pattern generating over 14K lines of code. Based on their ability to replicate the design evolution of the two real world applications, we continue to believe that the design evolution of object-oriented applications can greatly benefit from refactorings. The results for both real world applications are argued to be both general and scalable (Section 6.3.2 and Section 6.4.2).

Reduced Testing. Testing is one of the most significant costs resulting from the evolution of a design. In principle, refactorings can reduce testing because they are behavior-preserving. Initially, however, we would expect users to run a full slate of tests on refactored code given that there is no proof of behavior preservation. Successful applications of refactorings will lead to trust in their design and implementation. This is analogous to our trust in compiler technology: we believe that compilers will safely transform our code to assembly despite the fact that there is no proof of behavior preservation.

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

Ease of Exploration. Refactorings allow designers to experiment with new designs. While schema transformations and patterns are manually coded into applications today, it is clear that automating their introduction will allow

designers to more easily explore a design space without major commitments in coding and debugging time. This is analogous to the benefit gained from the introduction of WYSIWYG GUI editors. GUI editors transitioned user interface development from a time consuming batch job to an interactive point-and-click task. Similarly, refactorings have the potential to transition the batch-oriented design evolution process to an interactive point-and-click task. Ultimately, it may be the ability to evolve and explore new designs that will attract designers to this technology.

8.2 Refactoring limitations

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

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

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

`#define <constant> n` — in some cases these declarations can be replaced by a

statement of the form ‘`const <type> <constant> n`’ or a list of `#define`’s can be replaced by an enumerated type. Otherwise, this preprocessor information cannot be maintained.

`#define macro(x) F(x)` — function-like macros can sometimes be replaced by inline functions, however, macros which use the `#` and `##` string substitution operators may require changes to all sites which call the macro.

`#define FLAG` and `#ifdef` — one possible way to retain code that would be removed by `#ifdef` statements is to store it as a comment which is later uncommented after the source has been refactored. This solution would allow a program to be transformed correctly given one set of compiler flags but it could not guarantee correctness for a different set of flags.

A preferable solution may be for a refactoring to support sets of compilation flags for which behavior should be preserved. Besides introducing the difficulty of maintaining behavior across multiple versions of an application, this solution can suffer from exponential complexity given that for n flags, there are 2^n possible sets of flags for which behavior may need to be preserved.

We found that while much of the preprocessor information can be dealt with automatically, it is generally not possible to handle all cases that arise in large software applications.[1]

Computer formatting. Source code is parsed into an intermediate representation, transformed, and unparsed to produce a new version of the source code. In our implementation, original formatting information is not preserved in the intermediate representation, thus, all formatting is computer generated. Many programmers regard computer formatting to be undesirable. Recent tools demonstrate that it is possible to preserve a source’s original formatting

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

information, however, newly generated code must necessarily be computer formatted [Bax97[2], Bat98]. This is one possible disadvantage of automated program transformations over hand-coding.

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 presented in Chapter 4. The alternative is non-conservative conditions which may be difficult to design and implement or may require manual verification.

Automated Verification of Enabling Conditions. Some enabling conditions such as those ensuring that a program is not affected by object size or layout are verified manually (Section 3.4). Size and layout were not issues with the applications transformed in our experiments, however, users of refactorings must be aware of this limitation[3].

Since it is possible to design very conservative enabling conditions to guard against dependencies on object size and layout changes, a solution may be to implement these conservative conditions and allow the refactoring user to check the program manually if the conditions are violated.

Behavior Preservation. The AUIS example presented in Section 6.4 established that preservation of Opdyke's invariants is not sufficient to guarantee preservation of behavior. In light of this discovery, we take the following position on the important issue of behavior preservation:

---

2. Based on personal communication with Baxter.
3. Many applications use binary I/O which is often subject to size and layout constraints. This code must be carefully examined before being refactored.

- 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.
- Given a mature refactoring implementation, refactorings should be treated as trusted tools much as compilers are trusted to transform source code to assembly even when there is no mathematical proof to guarantee their correctness.
- We believe that the development of this trust will take time. The initial expectation is that refactoring users will retest transformed software to ensure that no errors are introduced. As refactorings continue to perform reliably, trust will grow and the benefit of reduced testing can be fully realized.

8.3 Refactoring requirements

Roberts identified three requirements for a Smalltalk refactoring implementation [Rob97]:

Integration. Refactorings must be integrated into the standard development tools. In Smalltalk, the standard development tool is the browser. Initially, Roberts added menu items to the browser for each refactoring. Eventually they implemented an entirely new browser. The integration of refactorings with other tools is discussed further in Section 9.2.

Efficiency. Refactorings must be fast. Smalltalk programmers are used to being able to immediately see the results of a change. Roberts argues that refactorings which are slow will not be used since Smalltalk programmers are likely to make the change by hand and live with the consequences. Based on our AUIS example which generated thousands of lines of C++, we believe that the specification of refactorings to be performed should be much faster than the time

required to manually code changes and that the elapsed time for executing refactorings is less important. Reduced testing of refactoring changes should also be taken into account to even further reduce the importance of refactoring execution times. An efficient representation for handling source-to-source transformations is discussed by Griswold [Gri91].

Naming. In Smalltalk, names are very important since that is one of the fundamental ways of determining what a class, method, or variable is used for. Reorganization tools that have to create entities often name them. These names do not have any meaning in the problem domain and serve to obfuscate the code. Roberts argues that whenever something is named, the user should always be prompted. We believe that default naming should be provided for many design pattern transformations for which classes are named after their roles, however, the user should be able to override any defaults.

Based on our refactoring implementation and experiments, we identify the following additional requirements:

Source File Access. Checks must be made to determine if any files affected by a transformation are read-only.

Code Placement. For C++, there may be multiple code placement options. The ability to place code has two important implications. First, if code is placed within an existing file, refactorings must support some method of selecting a position within a file where the new code should be placed. Although this could be done by specifying a file and a line number, ease of use dictates some form of visual interface. Second, choice of placing code in a new file implies that refactorings must have knowledge about the file structure and makefiles.

Makefiles. Refactorings intended for use in mainstream C++ development environments must accept a makefile or its equivalent as an argument. Makefiles define the set of files for a target application and specify what compilation flags

are set. Makefile compatibility minimizes the cost of integrating refactorings into a development environment.

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

Chapter 9

Conclusions

Design 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 designs evolve. These patterns can be recognized 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.

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 reflected the latest design.

We believe that a similar advance can occur for evolving object-oriented designs. Editing a design can be as simple as adding a line on a class diagram to represent an inheritance relationship or moving a variable from a subclass to a superclass. However, such changes must now be accompanied by painstakingly identifying lines of affected source code, manually updating the source, testing

the changes, fixing bugs, and retesting the application until the risk of new errors is sufficiently low. Furthermore, designs can require a great deal of experimentation. Multiple iterations of the design-implement-test cycle may be required to achieve a satisfactory final design.

Just as GUI editors revolutionized GUI design, we believe that refactoring-powered class diagram editors (where changes to an application's diagram automatically trigger corresponding changes to its underlying source code) may one day revolutionize the evolution of software designs.

9.1 Contributions

Our work makes the following contributions:

- **Automating design evolution.** Three kinds of design evolution in object-oriented systems are: schema transformations, the introduction of design pattern microarchitectures, and the hot-spot-driven-approach. All three can be viewed as transformations applied to an evolving design. Opdyke proposed refactorings for automating schema transformations. We extended the scope of what is automatable to include additional schema transformations, design patterns and hot-spot meta patterns. Our support of patterns in noteworthy because patterns are recognized as reusable elements of design. Based on the broad scope of changes supported, we believe that the majority of all object-oriented applications undergoes some form of automatable evolution.
- **Refactorings.** We designed and implemented a set of refactorings which automate the suite of schema transformations, design patterns, and hot-spot meta patterns identified in Chapter 4. Refactorings were implemented as command line executables which could be integrated into the majority of all C++ development environments. We added to Opdyke's list of invariants for preserving behavior but note that there is no formal proof that the list of

invariants identified is complete. We argued that refactorings preserve behavior because of good engineering and not because of any mathematical guarantee.

- **Scalability.** We showed that general-purpose refactorings can automate significant design changes on large, real-world applications. The changes automated involved thousands of lines of code previously modified by hand.
- **Benefits, limitations, and requirements.** Our research clearly showed the benefits that could result from a refactoring tool. Refactorings can reduce the cost of evolutionary maintenance by performing design changes that would otherwise be tedious or error-prone to accomplish by hand. Refactorings also reduce the need for overly complex designs and facilitate exploration of the space of possible designs.

  Our experiments also revealed requirements and limitations that must be acknowledged before refactoring technology can be transitioned beyond academic prototypes. A refactoring tool must preserve comments, handle certain preprocessor directives (e.g., constant declarations and inlined functions), have the ability to read makefiles (or their equivalent) to understand the set of source files that are to be transformed, provide options on code placement, and deal with restricted access to source code. There are limitations that users of a refactoring tool may need to observe: users must work with computer formatted code, some preprocessor directives (e.g., function-like macros which use ##) cannot be supported, refactorings often have conservative enabling conditions and some enabling conditions must be verified manually.

9.2 Future directions

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.

Granularity of transformations. Our research proposes a basis set of primitive refactorings. Larger grain refactorings up to the size of design patterns are likely to be more convenient in practice. In both experiments, the number of refactorings could be significantly reduced with larger grain refactorings (Section 6.3.2 and Section 6.4.2).

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

Program families. Transformation systems must recognize that many files may be included by multiple programs. When transforming a file used by more than one program, it is desirable for the transformations system to check enabling conditions for all programs which use that file. Otherwise, a file might be transformed safely for one program while causing another program which uses the same file to break.

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

Integration with other tools. Refactorings packaged as individual executables which take a makefile target as an argument are not dependent on the presence of other tools. In this form, refactorings can be integrated into most

mainstream development environments because most environments support command-line access to source code.

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

Java as a target language. 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.[1] Tools are now being developed to aid in this process [Sim95, Bax97, Bat98].

1. When we began our work, the future for Java appeared uncertain, tool support was not available, and large Java applications were nonexistent. This is no longer true today.

Appendix A: Refactorings

This appendix defines the new refactorings contributed by our research (refactorings in italics from the list in Chapter 3, Table 1). Invariants preserved by each enabling condition are noted in parenthesis. Enabling conditions which were verified by hand are denoted with a (*)[1].

1. Note, however, that it is possible to automate conservative checks for any of the enabling conditions. Thus, the decision of which checks to automate and the conservatism of each refactoring is implementation dependent.

Add factory method

Name:

add_factory_method[Product, Ptype, Factory, method]

Purpose:

To create a factory method for a class.

Arguments:

Product - the class of the object which is created

Ptype - the return type of the factory method. **Ptype** must be **Product** or a superclass of **Product**.

Factory - the class to which the factory method is added

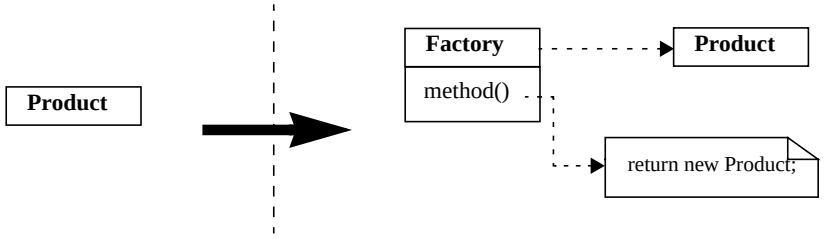
`method()` - the name of the method added

Description:

add_factory_method adds a class `method()` to **Factory**. `method()` returns a **Product** object and has a return type of **Ptype**. Occurrences of "`new Product`" are replaced by a call to `Factory::method()`.

Enabling Conditions:

1. `method()` must not clash with a method of **Factory**.



(a) Initial State **(b) Target state**

Add variable

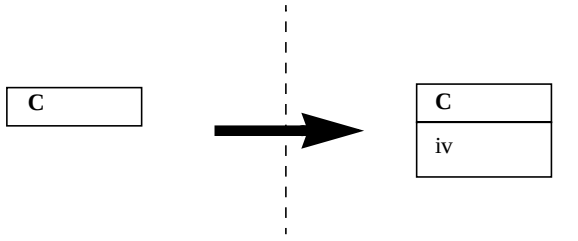
Name:
add_variable[C, iv, type, access, init_expression]

Purpose:
To add an instance variable to a class.

Arguments:
C - the class to which the variable is added
`iv` - the new instance variable
`type` - the type of the variable
access - PUBLIC, PRIVATE, or PROTECTED[1]
<init_expression> - a legal C++ expression used in **C**'s constructor to initialize `iv`

Description:
add_variable adds instance variable `iv` to class **C**.

Enabling Conditions:
1. `iv` must not clash with an existing member or global variable.
2. If `type` is a class, then it cannot be pure virtual.
3. If access is PRIVATE or PROTECTED, then **C** cannot be initialized with initializer lists.
4. <init_expression> cannot have any side effects when evaluated to initialize `iv`.*
5. If `type` is a class, then its constructor cannot have any side effects beyond initializing the object created.*
6. Program behavior must not depend on the size or layout of **C**.*



(a) Initial State **(b) Target state**

1. We describe a general add_variable which can add a public, private, or protected instance variable, however, our implementation always adds a public variable.

Composite

Name:

composite[C, iv, method]

Purpose:

To add a method which forwards messages through a chain of objects.

Arguments:

C - the class to which `iv` and `method()` are added

`iv` - the name of the instance variable to be created. `iv` is a collection of **C** objects.

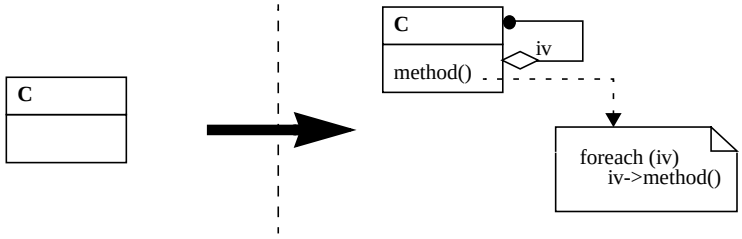
`method()` - the method to be created that forwards messages through all objects in `iv`

Description:

composite adds `iv` and `method()` to class **C**. `method()` forwards a message to all objects in `iv`. Subclasses of **C** may call `method()` to forward a message to objects stored in `iv`. This refactoring is based on the Composite design pattern [Gam95].

Enabling Conditions:

1. `iv` must not clash with an existing member or global variable.
2. `method()` must not clash with a method of **C**.
3. Program behavior must not depend on the size or layout of **C**.*



(a) Initial State **(b) Target state**

Create iterator

Name:

create_iterator[C, List, ListIterator]

Purpose:

To provide classes for storing a list of objects and for accessing this list.

Arguments:

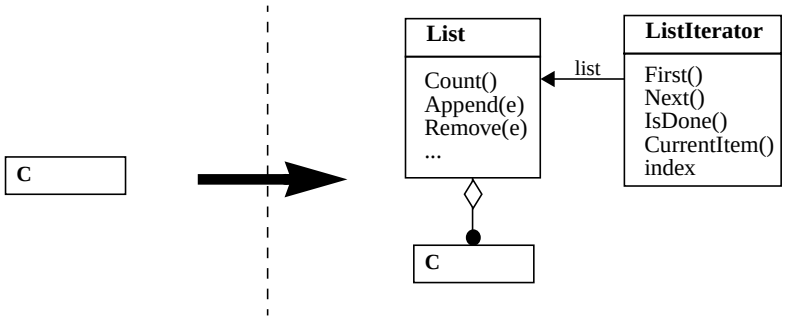
C - the class whose objects are being stored in a list
List - a list of **C** objects
ListIterator - a class for traversing the elements of a **List**

Description:

create_iterator creates a two classes. The first is a List class which stores a list of C objects. The second is the ListIterator class which is used to traverse List. Separation of the iterator from the list class allows different iterators to traverse the list in multiple ways. This refactoring is based on the Iterator design pattern [Gam95].[1]

Enabling Conditions:

1. **List** must be a unique class.
2. **ListIterator** must be a unique class.



(a) Initial State **(b) Target state**

1. Note that multiple implementations of the list and iterator classes are possible. Budinsky explores this issue in [Bud96].

Create method accessor

Name:

create_method_accessor[C, iv, iv_method, accessor]

Purpose:

To create an accessor which replaces a method called through an instance variable.

Arguments:

C - the class containing `iv`

`iv` - an instance variable whose type is an object pointer.

`iv_method` - a method which must be supported by the object pointed to by `iv`

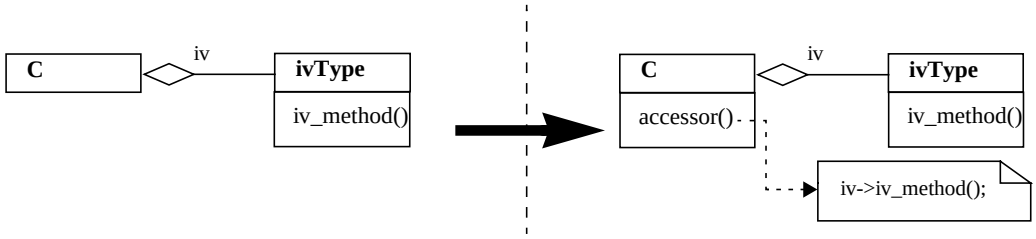
`accessor` - a method which replaces a call to `iv->iv_method()`

Description:

create_method_accessor creates the method `accessor()` in class **C** which replaces all calls to `iv->iv_method()`.

Enabling Conditions:

1. `accessor` must not clash with existing functions.
2. Program behavior must not depend on the size or layout of **C**.*



(a) Initial State **(b) Target state**

Declare abstract method

Name:

declare_abstract_method[*Base*, Derived, method(type_signature)]

Purpose:

To declare a method defined in a derived class in a base class.

Arguments:

Base - the class in which the method will be declared

Derived - the class in which the method will be declared

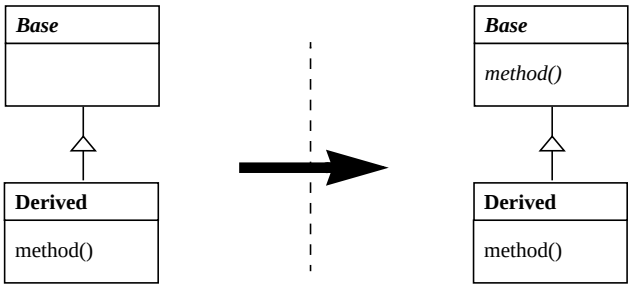
`method(type_signature)` - the method to be declared. The type signature uniquely identifies the method since the same method name can have multiple type signatures.[1]

Description:

declare_abstract_method declares `method()` implemented in class **Derived** as a pure-virtual function in ***Base***.

Enabling Conditions:

1. All subclasses of ***Base*** must support `method()`.
2. `method()` must not clash with an existing method in ***Base***.
3. ***Base*** cannot be initialized with an initializer list.
4. The program cannot instantiate ***Base***.



(a) Initial State **(b) Target state**

1. Type signature matching was not implemented because the applications we evolved allowed us to assume that the methods could be uniquely identified without specifying the type signature.

Decorator

Name:

decorator[C, iv, method]

Purpose:

To add a method which forwards messages to a collection of objects.

Arguments:

C - the class to which `iv` and `method()` are added

`iv` - the name of the instance variable to be created. `iv` is a pointer to a **C** object.

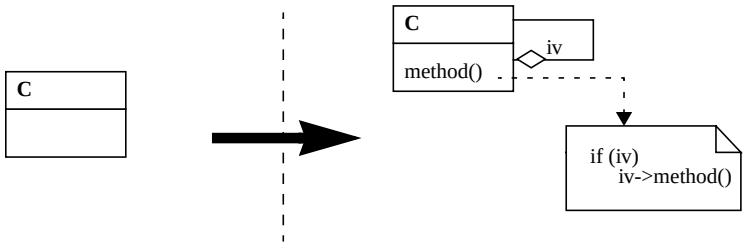
`method()` - the method to be created that forwards messages through `iv`

Description:

decorator adds `iv` and `method()` to class **C**. `method()` calls itself through `iv` whenever `iv` is non-null. Subclasses of **C** may call `method()` to forward a message through a chain of objects linked through `iv`. This refactoring is based on the Decorator design pattern [Gam95].

Enabling Conditions:

1. `iv` must not clash with an existing member or global variable.
2. `method()` must not clash with a method of **C**.
3. Program behavior must not depend on the size or layout of **C**.*



(a) Initial State **(b) Target state**

Inherit

Name:

inherit[*Base*, Derived]

Purpose:

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

Arguments:

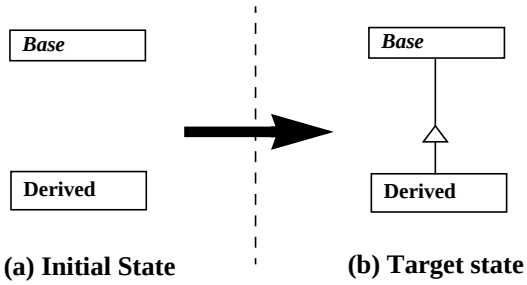
Base `- superclass name`
Derived `- subclass name`

Description:

inherit makes ***Base*** a superclass of **Derived**.

Enabling Conditions:

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**.



(a) Initial State **(b) Target state**

Move variable across object boundary

Name:

move_variable_across_object_boundary[C, iv, iv_target]

Purpose:

To move an instance variable of a class to one of its components.

Arguments:

C - the class containing the variable to be moved

`iv` - the instance variable to be moved

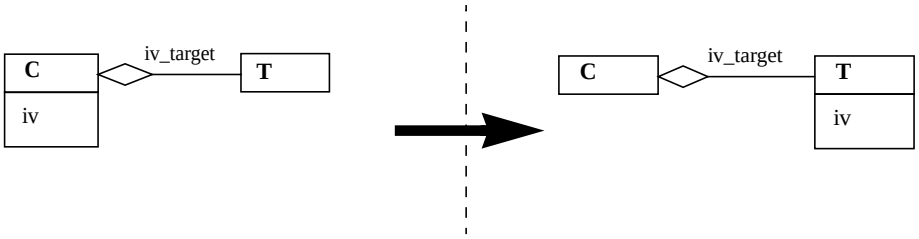
`iv_target` - `iv` is moved to the object stored in this variable. This variable must be an object and not an object pointer.

Description:

Given `iv_target` is of class **T**, **move_variable_across_object_boundary** moves instance variable `iv` to class **T**. `iv` is then accessed through `iv_target`.

Enabling Conditions:

1. `iv` must not clash with an existing member or global variable in **T**.
2. The constructor of **T** cannot have any side effects beyond initializing the object created.*
3. Program behavior must not depend on the size or layout of **C** and **T**.*



(a) Initial State **(b) Target state**

Procedure to method

Name:

procedure_to_method[proc(type_signature), arg]

Purpose:

To convert a procedure which takes an object as an argument to a method on that object.

Arguments:

`proc(type_signature)` - a procedure uniquely identified by its type signature

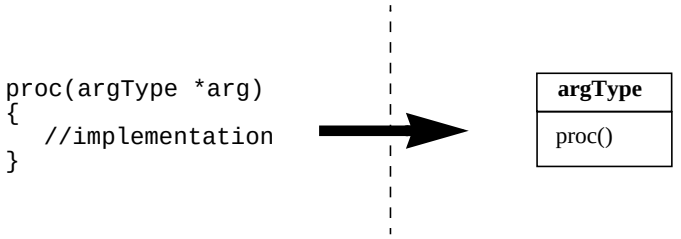
`arg` - an argument of `proc()` whose type is a pointer to an **argType** object. **argType** must be a class.

Description:

procedure_to_method converts `proc()` to a method of **argType**. `arg` is removed from the type signature of `proc()` and references to `arg` within `proc()` are converted to references to `this`. Calls to `proc()` are converted to invocations of the method on `arg`.

Enabling Conditions:

1. `proc()` must not clash with a method of **argType**.
2. The program must not use `proc` as a function pointer.



(a) Initial State **(b) Target state**

Procedure pointer to command

Name:

procedure_ptr_to_command[C, iv, Command, ((procedure1, Command1), (procedure2, Command2), ...)]

Purpose:

To convert the type of a variable from a function pointer to a Command class pointer.

Arguments:

C - the class containing `iv`

`iv` - an instance variable whose type is to be converted from a function pointer to a **Command** pointer

Command - the base class for all commands

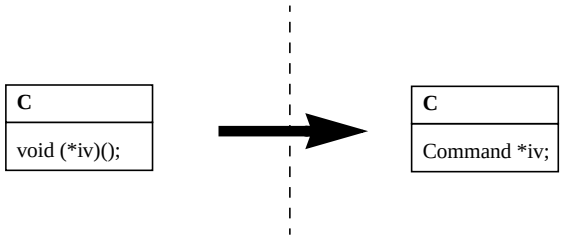
(`procedure1`, **Command1**) - an association list of function pointers and their corresponding commands

Description:

procedure_ptr_to_command converts the type of `iv` from a function pointer to a **Command** pointer. Function pointers assigned to `iv` are replaced by their corresponding commands specified in the association list of (procedure, command) pairs. Expressions which dereference `iv` are converted to calls to `iv`'s `Execute()` method.[1]

Enabling Conditions:

1. All assignments must be type safe[2].
2. Procedures and their corresponding commands must be equivalent.*



(a) Initial State **(b) Target state**

1. See Section 6.4.1 for code-level example of changes.
2. This is actually a postcondition which can be tested by a compiler check for "type mismatch" errors. An example violation would occur in a program in which an automatic variable is initialized by Cobj.iv. When iv is converted to a Command, the variable initialization would flag a type mismatch. It is possible to write a refactoring which would also convert the types of automatic variables and procedure arguments.

Procedure to command

Name:

procedure_to_command[ProcName, Command]

Purpose:

To objectify a procedure.

Arguments:

`ProcName()` - the procedure to be converted to a command

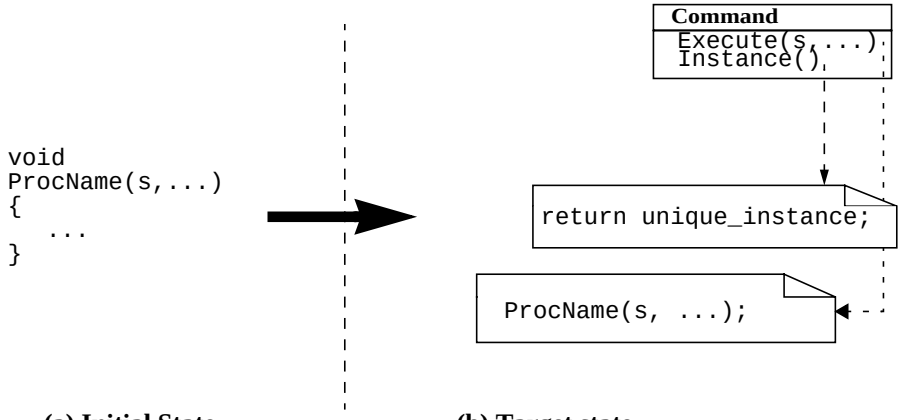
Command - the name of the command class to be created

Description:

procedure_to_command converts `ProcName()` from a procedure to a **Command** class [Gam95]. The `Execute()` method executes `ProcName()`. The arguments to `Execute()` and `ProcName()` are identical allowing calls to `ProcName` through pointer dereferencing to be replaced by calls to `Execute()`. To preserve a function pointer's identity, the **Command** class is created as a Singleton [Gam95]. The `Instance()` method returns the single instantiation of **Command** class.[1]

Enabling Conditions:

1. **Command** must be a unique class.



(a) Initial State **(b) Target state**

1. Another option would have been to combine procedure_to_command which creates commands and procedure_ptr_to_command which performs the type change, into a single refactoring. This would eliminate the need to check if function pointers and commands matched since the commands would be created directly from the function pointers. This is an example where it is easier to verify preconditions for a larger grain refactoring rather than for a series of equivalent lower-level refactorings.

Singleton

Name:

singleton[C]

Purpose:

To create a class that has only one instance.

Arguments:

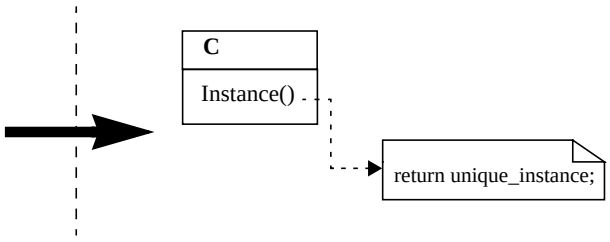
C - the name of the singleton class to be created

Description:

singleton creates a class **C** that has only one instance. The `Instance()` method returns the single instantiation of **C**. This refactoring and its source level implementation are based on the Singleton design pattern [Gam95].

Enabling Conditions:

1. **C** must be a unique class.



(a) Initial State **(b) Target state**

Structure to pointer

Name:

structure_to_pointer[C, iv]

Purpose:

To convert an instance variable from a class structure to an object pointer.

Arguments:

C - the class containing `iv`

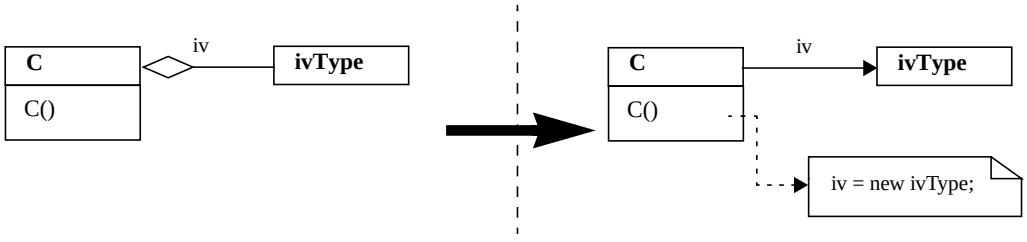
`iv` - the instance variable of type **ivType** which is converted to a pointer to **ivType**

Description:

structure_to_pointer converts `iv` from an **ivType** to a pointer to an **ivType**. All references to `iv` are also converted. `iv` is initialized to point to a new **ivType** object in **C**'s constructor.

Enabling Conditions:

1. **C** cannot be initialized with an initializer list.
2. **ivType**'s constructor cannot have any side effects beyond initializing the object created.*
3. Program behavior must not depend on the size or layout of **C**.*



(a) Initial State **(b) Target state**

Structure to class

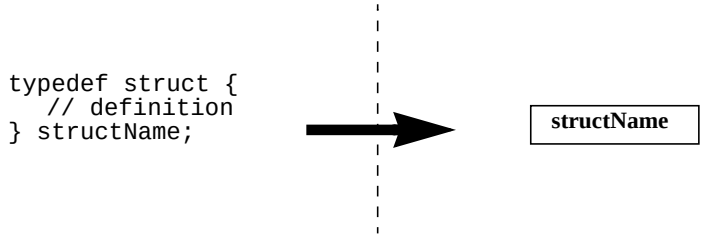
Name:
structure_to_class[structName]

Purpose:
To convert a structure to a class.

Arguments:
structName - a C++ structure

Description:
structure_to_class converts `struct` to a class called **structName**.

Enabling Conditions:
None



(a) Initial State **(b) Target state**

Substitute

Name:

substitute[C, *Base*, Derived]

Purpose:

To change a classes dependence on a derived class to a dependence on its base class.

Arguments:

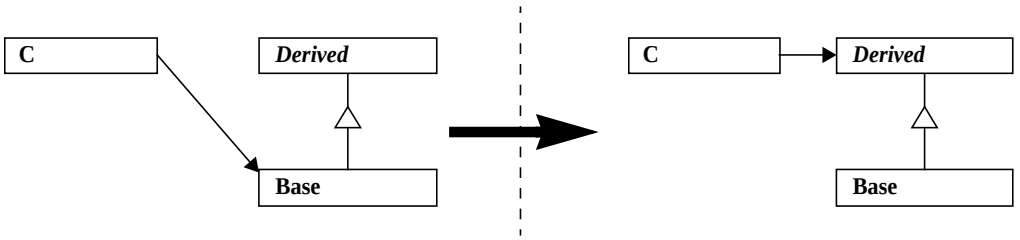
C - the class in which the substitution occurs
Base - the class being inserted
Derived - the class being replaced

Description:

substitute substitutes class **C**'s references to class **Derived** with references to class ***Base***. All structures and pointers in class **C** including instance variables, method return types, method arguments, method local variables, etc. are converted.

Enabling Conditions:

1. ***Base*** must support **Derived**'s interface to **C**. Automated checking of this condition is equivalent to performing the substitution, recompiling, and checking for 'undefined variable' or 'undefined method' errors. **Derived** cannot override any part of this interface.
2. Assignment, argument passing, and return values involving **Derived** objects must accept objects of type ***Base***. Automated checking of this condition is equivalent to performing the substitution, recompiling, and checking for 'type mismatch' errors.
3. If **Derived** is instantiated, then the constructors for ***Base*** and **Derived** must be semantically equivalent.*
4. If **Derived** is instantiated, then ***Base*** cannot be pure-virtual.
5. Program behavior must not depend on the size or layout of **Derived**.*



(a) Initial State **(b) Target state**

Appendix B: Supported patterns

This appendix provides derivations for the additional design patterns and hot-spot meta patterns whose introduction can be automated with refactorings.

B.1 Design patterns

B.1.1 Builder

Builder separates the construction of a complex object from its representation so that the same construction process can create different representations. In the example from [Gam95], the rich text format reader class **RTFReader** points to a builder class which supports conversion to different text formats (only the ASCII converter class is displayed in Figure B.1). A design originally supporting a single format (Figure B.2) can be transformed to use the builder design pattern in five steps:

1. *Create the builder classes.* Create the **ASCIIConverter** and ***TextConverter*** classes using **create_class**. Make **ASCIIConverter** a subclass of ***TextConverter*** using **inherit** (Figure B.3).
2. *Link the original class to the new builder class.* Add instance variable `builder` to **RTFReader** using **add_variable** (Figure B.4).
3. *Move instance variables and methods to the builder class.* In this example,

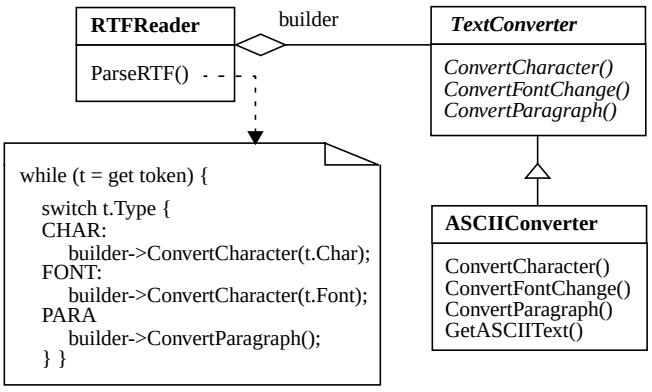


Figure B.1: Builder design pattern example

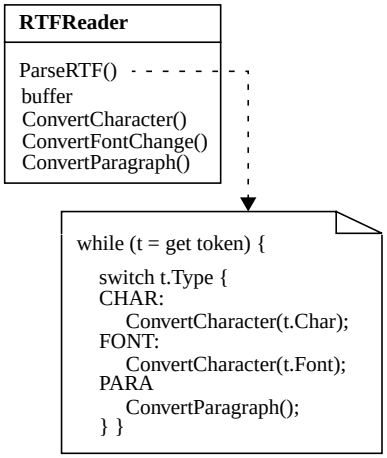


Figure B.2: Design for a single text format

the original design stores converted text in **RTFReader**'s `buffer` instance variable. Move `buffer` to **ASCIIConverter** using **move_variable_across_object_boundary**.[1] Create an accessor for the buffer instance variable called `GetASCIIText()` using **create_variable_accessor**. **Create_variable_accessor** replaces direct access to an instance variable

1. The instance variable "buffer" and the methods ConvertCharacter(), ConvertFontChange(), and ConvertParagraph() are not a part of the public interface of RTFReader in the example from [Gam95]. Thus, no accessors are created when the members are moved to ASCIIConverter.

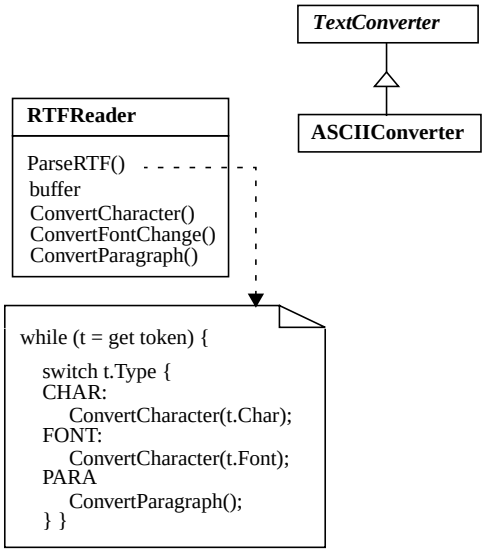


Figure B.3: Builder classes created

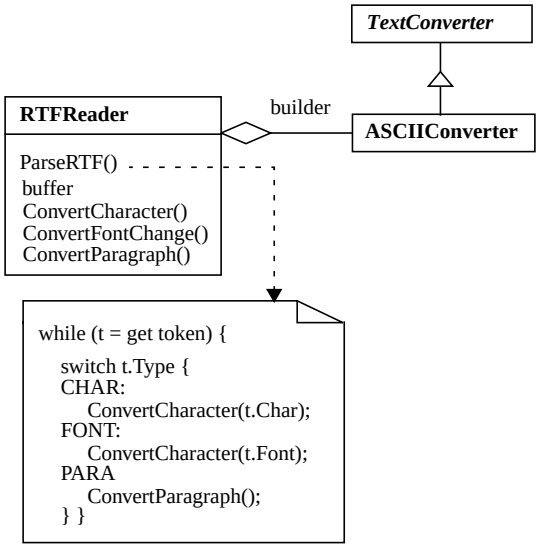


Figure B.4: Builder instance variable added to RTFReader

with methods created to get and set the variable. Move methods `ConvertCharacter()`, `ConvertFontChange()`, and `ConvertParagraph()` to **ASCIIConverter** using

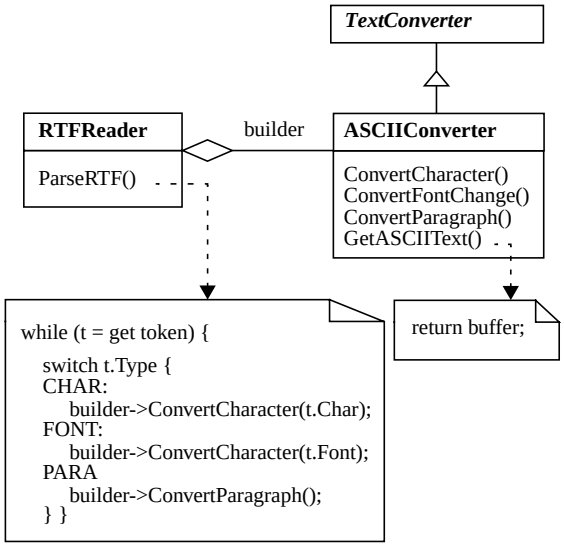


Figure B.5: Converter methods moved to XWindow class

move_method_across_object_boundary (Figure B.5).

4. *Declare methods in the builder superclass.* Declare `ConvertCharacter()`, `ConvertFontChange()`, and `ConvertParagraph()` in ***TextConverter*** using **declare_virtual_method** (Figure B.6).
5. *Generalize the original class to accept any builder class.* Change the type of the `builder` instance variable from **ASCIIConverter** to ***TextConverter*** using **substitute** (Figure B.1).

B.1.2 Strategy

Strategy defines a family of algorithms, encapsulates each one, and makes them interchangeable. In the example from [Gam95], a **Composition** class is responsible for maintaining and updating the linebreaks of text displayed in a text viewer (Figure B.7). Linebreaking strategies are implemented in subclasses of ***Compositor.*** A **Composition** maintains a reference to a **Compositor** and forwards the linebreaking

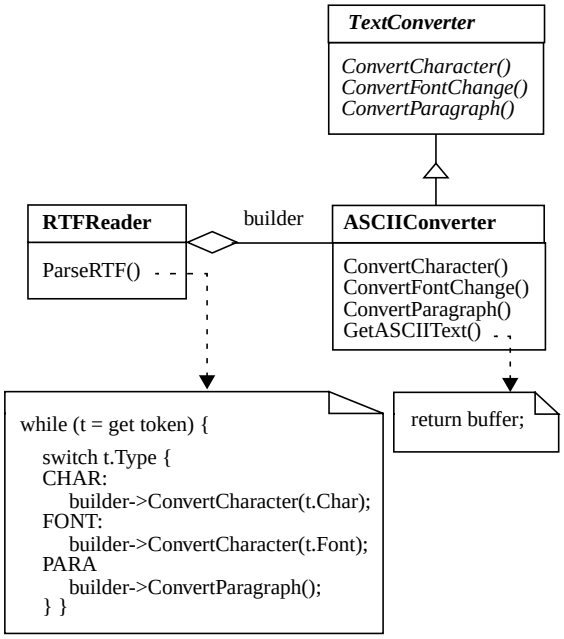


Figure B.6: Virtual methods declared in TextConverter

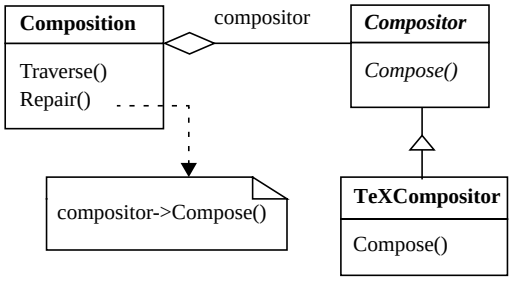


Figure B.7: Strategy design pattern example

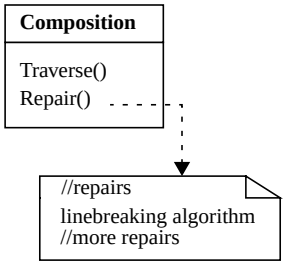


Figure B.8: Strategy for a single linebreaking algorithm

responsibility to this class. A design supporting a single linebreaking strategy (Figure B.8) can be transformed to use the strategy design pattern (Figure 7) in six steps:

1. *Encapsulate the strategy as a method.* Replace the linebreaking algorithm with a `Compose()` method using **extract_code_as_method** (Figure B.9). **Extract_code_as_method** replaces a block of code with a function call

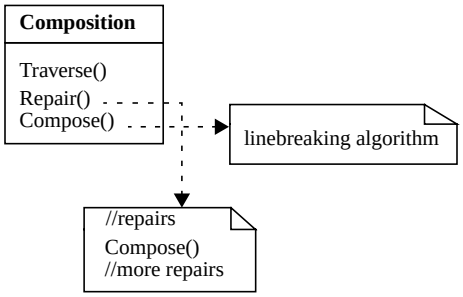


Figure B.9: Linebreaking algorithm extracted as a method

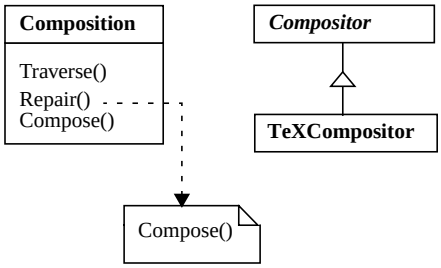


Figure B.10: Strategy classes created

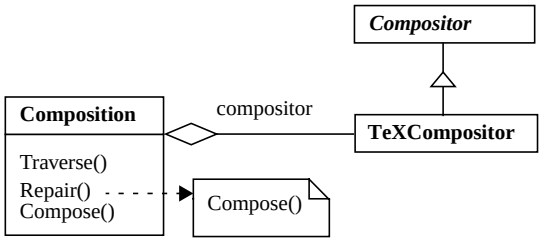


Figure B.11: Strategy instance variable added to Composition

which executes the block.

2. *Create the strategy classes.* Create the ***Compositor*** and **TeXCompositor** classes using **create_class**. Make **TeXCompositor** a subclass of ***Compositor*** using **inherit** (Figure B.10).
3. *Link the original class to the strategy class.* Add the `compositor` instance variable using **add_variable** (Figure B.11).
4. *Move the strategy method to the strategy class.* Move the `Compose()` method to **TeXCompositor** using **move_variable_across_object_boundary** (Figure B.12).
5. *Declare the strategy method in the strategy superclass.* Declare `Compose()` in **Compositor** using **declare_virtual_method**.

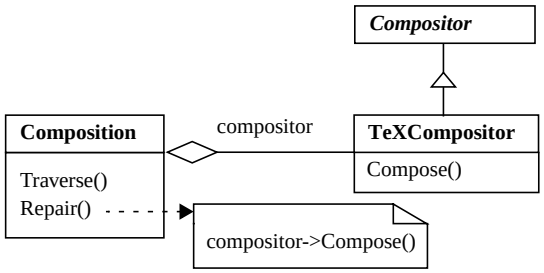


Figure B.12: Compose() method moved to strategy class

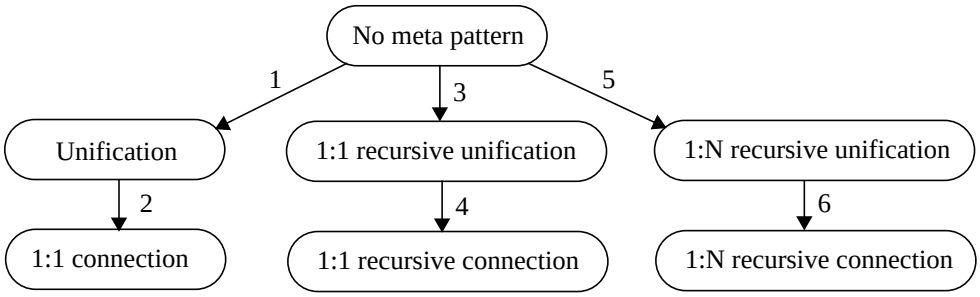


Figure B.13: Hot-spot meta pattern transitions enabled by refactorings

6. *Generalize the original class to accept any strategy class.* Change the type of the `compositor` instance variable from **TeXCompositor** to ***Compositor*** using **substitute** (Figure B.7).

B.2 Hot-spot meta patterns

For convenience, the meta pattern transition diagram from Section 4.3.2 is presented again in Figure B.13. Derivations for transitions 1 and 2 were given in Section 4.3.2. Derivations for the remaining transitions follow.

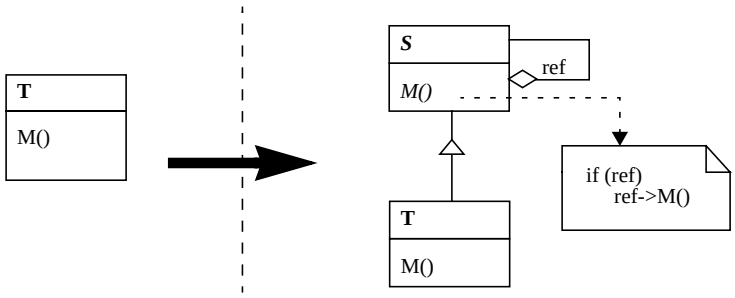


Figure B.14: Class and method to 1:1 recursive unification

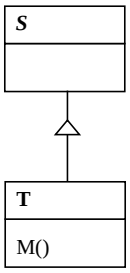


Figure B.15: Superclass S created

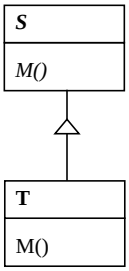


Figure B.16: Method M() declared in S

B.2.1 No meta pattern to 1:1 recursive unification

Figure B.14 displays the transformation of a class and method with no recursion to use the 1:1 recursive unification composition (transition 3 from Figure B.13). In 1:1 recursive unification, there is one template-hook method which calls itself recursively by default (Figure B.14, right hand side). Subclasses containing methods which override the template-hook method can call the superclass method to forward messages along a chain of objects. The 1:1 recursive unification meta pattern can be added in three steps:

1. *Create a superclass of the template-hook class.* Create class ***s*** using **create_class**. Make ***s*** a superclass of `T` using **inherit** (Figure B.15).
2. *Declare the template-hook method in the superclass.* Declare `M()` in **s**

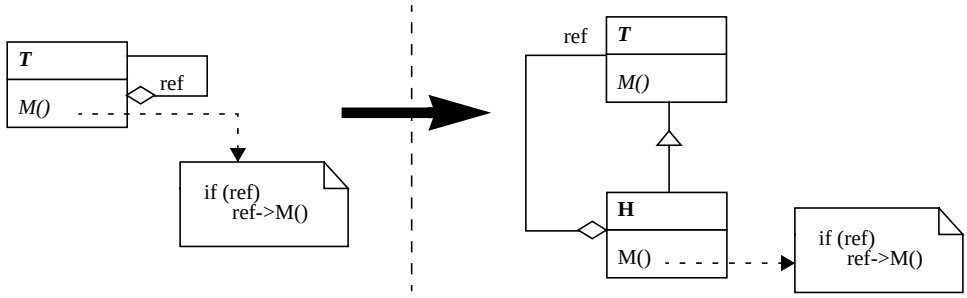


Figure B.17: 1:1 recursive unification to 1:1 recursive connection

using **declare_virtual_method** (Figure B.16).

3. *Add the recursive template-hook method to the superclass.* Apply the **decorator** refactoring to ***S*** and `M()`(Figure B.14, right hand side). **Decorator** adds instance variable `ref` of type ***S*** and creates a method `M()` which forwards the message to `ref`. Subclasses of ***S*** which override `M()` can then call `S::M()` to pass messages down a chain.

B.2.2 1:1 recursive unification to 1:1 recursive connection

Figure B.17 displays the transition from 1:1 recursive unification to 1:1 recursive connection (transition 4 in Figure B.13). In 1:1 recursive connection (Figure B.17, right hand side), the template and hook methods have the same name. An object of type ***T*** can be a single object or a chain of objects of subtype **H**. The transformation from 1:1 recursive unification to 1:1 recursive connection can be accomplished in two steps:

1. *Create the hook class as a subclass of the template class.* Create class **H** using **create_class**. Make ***T*** a superclass of **H** using **inherit** (Figure B.18).
2. *Move methods and variables to the hook class.* Move `M()` from ***T*** to **H** using **push_down_method**. Declare `M()` in ***T*** using **declare_virtual_method** (Figure B.19). Move `ref` from ***T*** to **H** using **push_down_variable** (Figure B.17, right hand side).

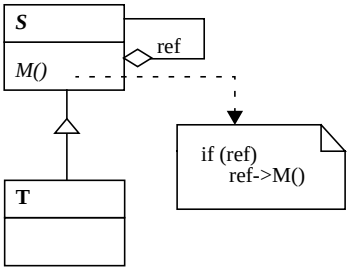


Figure B.18: Superclass S created

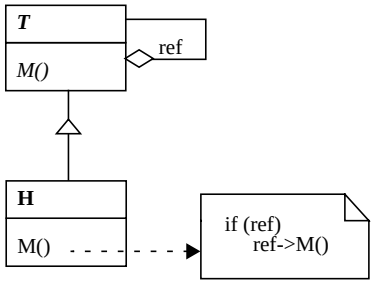


Figure B.19: Method M() declared in S

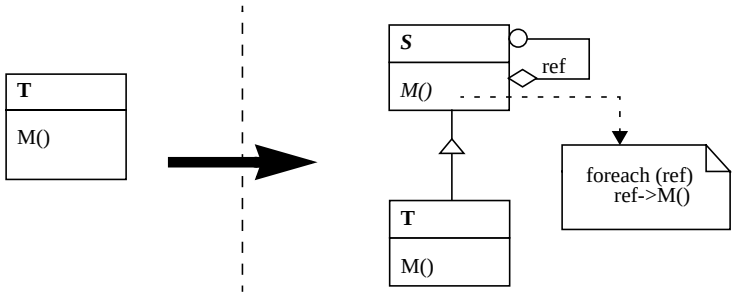


Figure B.20: Class and method to 1:N recursive unification

B.2.3 No meta pattern to 1:N recursive unification

Figure B.20 displays the transformation of a class and method with no recursion to use the 1:N recursive unification composition (transition 5 from Figure B.13). In 1:N recursive unification, there is one template-hook method which calls itself recursively by default (Figure B.20, right hand side). Subclasses containing methods which override the template-hook method can call the superclass method to forward messages along a tree of objects. The 1:1 recursive unification meta pattern can be added in three steps:

1. *Create a superclass of the template-hook class.* Create class ***s*** using **create_class**. Make ***s*** a superclass of **T** using **inherit** (Figure B.21).

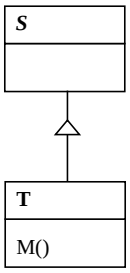


Figure B.21: Superclass S created

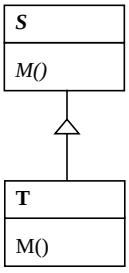


Figure B.22: Method M() declared in S

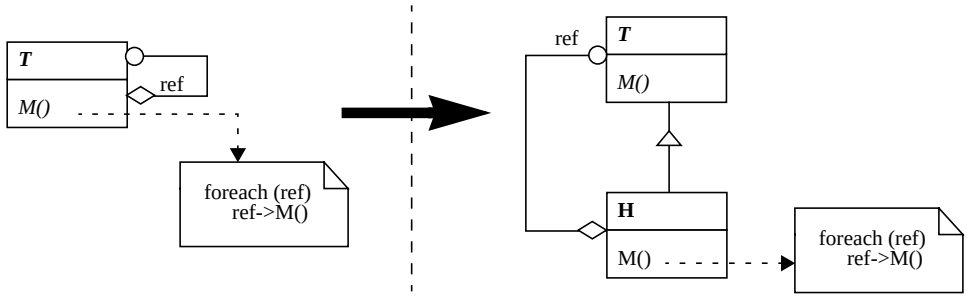


Figure B.23: 1:N recursive unification to 1:N recursive connection

2. *Declare the template-hook method in the superclass.* Declare `M()` in `S` using **declare_virtual_method** (Figure B.22).
3. *Add the default recursive template-hook method to the superclass.* Apply the **composite** refactoring to ***s*** and `M()`(Figure B.20, right hand side). **Composite** adds an instance variable `ref` which stores a collection of objects of type ***s*** and creates a method `M()` which forwards the message to all objects in the collection. Subclasses of ***s*** which override `M()` can then call `S::M()` to pass messages through a tree.

B.2.4 1:N recursive unification to 1:N recursive connection

Figure B.23 displays the transition from 1:N recursive unification to 1:N recursive connection (transition 6 in Figure B.13). In 1:N recursive connection, (Figure B.23, right hand side), the template and hook methods have the same name. An

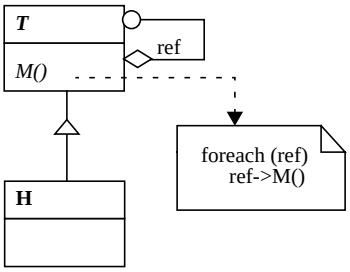


Figure B.24: Superclass S created

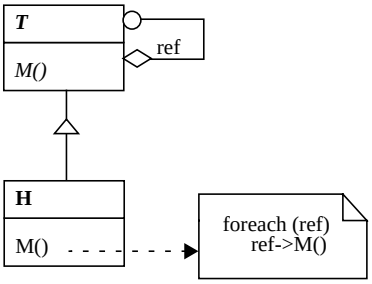


Figure B.25: Method M() declared in S

object of type ***T*** can be a single object or a tree of objects of subtype **H**. The transformation from 1:N recursive unification to 1:N recursive connection can be accomplished in two steps:

1. *Create a hook class as a subclass of the template class.* Create class **H** using **create_class**. Make ***T*** a superclass of **H** using **inherit** (Figure B.24).
2. *Move methods and variables from the template class to the hook class.* Move `M()` from ***T*** to **H** using **pull_down_method**. Declare `M()` in **T** using **declare_virtual_method** (Figure B.25). Move `ref` from ***T*** to **H** using **pull_down_variable** (Figure B.23, right hand side).

Bibliography

[And94] B. Anderson. Patterns: Building Blocks for Object-Oriented Software Architectures. In *Software Engineering Notes*, January 1994.

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

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

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

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

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

[Bec94] K. Beck, R. Johnson. Patterns Generate Architectures. In *Proceedings ECOOP '94*, Springer-Verlag, 1994.

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

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

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, California, 1994.

[Bos98] J. Bosch. Design Patterns as Language Constructs. In *Journal of Object-Oriented programming*, Vol. 11, No. 2, pp. 18-32, May 1998.

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

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

[Cha98] E. Chan, J. Boyland, and W. Scherlis. Promises: Limited Specifications for Analysis and Manipulation. In *Proceedings of ICSE '98*, 1998.

[Coa92] P. Coad. Object-Oriented Patterns. In *Communications of the ACM*, V35 N9, pages 152-159, September 1992.

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

[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.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, Massachusetts, 1984.

[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*. PhD thesis. University of Washington. August 1991.

[Gri93] W. Griswold. Direct Update of Data Flow Representations for a Meaning-Preserving Program Restructuring Tool. In *SIGSOFT ’93*, December 1993.

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

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

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

[Joh92] R. Johnson. Documenting Frameworks with Patterns. In *OOPSLA ’92 Proceedings*, SIGPLAN Notices, 27(10), pages 63-76, Vancouver BC, October 1992.

[Kem92] Chris F. Kemerer. How the Learning Curve Affects CASE Tool Adoption. In *IEEE Software*, pages 23-28, May 1992.

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

[Kra88] G. E. Krasner and S. T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. In *Journal of Object-Oriented Programming*, pages 26-49, August 1988.

[LaL91] W. LaLonde and J. Pugh. Subclassing != Subtyping != Is-a. In *Journal of Object-Oriented Programming*, pages 57-62, January 1991.

[Lie89] K. Lieberherr and I. M. Holland. Assuring good style for object-oriented programs. In *IEEE Software*, September 1989.

[Lie88] K. Lieberherr, I. M. Holland, and A. Riel. Object-oriented programming: An objective sense of style. In *Proceedings OOPSLA '88*, September 1988.

[Lie91] K. Lieberherr, W. Hursch, and C. Xiao. Object-Extending Class Transformations. Techinal report, College of Computer Science, Northeastern University, 360 Huntington Ave., Boston, Massachusetts, 1991.

[Lin92] M. Linton. Encapsulating a C++ Library. In *Proceedings of the 1992 USENIX C++ Conference*, pages 57-66, Portland, Oregon, August 1992.

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

[Mey88] Ware Meyers. Interview with Wilma Osborne. In IEEE Software 5 (3), pages 104-105, 1988.

[McG97] P. McGuire. Lessons Learned in the C++ Reference Development of the SEMATECH Computer-Integrated Manufacturing (CIM) Applications Framework. In *SPIE Proceedings*, Volume 2913, pages 326-344, 1997.

[Mor85] Morris, J. H., M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S.H. Rosenthal, and F.D. Smith. Andrew: A Distributed Personal Computing Environment. *Communications of the ACM*, March, 1986.

[Opd92] W. F. Opdyke. Refactoring Object-Oriented Frameworks. PhD 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.

[O'Sh86] Tim O'Shea, Kent Beck, Dan Halbert, and Kurt J. Schmucker. Panel on: The learnability of object-oriented programming systems. In *Proceedings of OOPSLA '86*. pages 502-504. November 1986.

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

[Pree95] 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.

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

[Rei94] S. Reiss. CPPP 1.61 (software package), Brown University, 1994. Available via anonymous ftp from *ftp.cs.brown.edu:/pub/cppp.tar.Z*.

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

[Roc86] R. Rochat. In search of good Smalltalk programming style. Technical Report CR-86-19, Tektronix, 1986.

[Rum91] J. Rumbaugh et. al. *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Sch98a] W. Scherlis. Systematic Change of Data Representation: Program Manipulations and Case Study. In *Proceedings of ESOP '98*, 1998.

[Sch98b] B. Schulz et. al. On the Computer Aided Introduction of Design Patterns into Object-Oriented Systems. In *Proceedings of the 27th TOOLS Conference*, IEEE CS Press, 1998.

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

[Ste95] S. Stewart. *Roadmap for the Computer-Integrated Manufacturing Application Framework*. NISTIR 5697, June, 1995.

[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.

[Tok99a] L. Tokuda and D. Batory. Automating Three Modes of Object-Oriented Software Evolution. In *Proceedings of COOTS '99*, May 1999.

[Tok99b] L. Tokuda and D. Batory. Evolving Object-Oriented Architectures with Refactorings. To appear in *ASE '99*.

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

[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.

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

Vita

Lance Tokuda was born in Kaneohe, Hawaii, the son of George and Janet Tokuda. After graduating from Castle High School, he entered the University of Hawaii at Manoa where he received a Bachelor of Science with Distinction in Electrical Engineering in August 1986, and Master of Science in Electrical Engineering in August 1987. Lance worked a year at ESL in Sunnyvale before being recruited as employee number four and principal engineer at Resumix Inc. In September of 1993, he entered the Graduate School of the University of Texas at Austin. A list of publications follows:

1. 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.
2. L. Tokuda and D. Batory. Automating Three Modes of Object-Oriented Software Evolution. In *Proceedings of COOTS '99*, May 1999.
3. L. Tokuda and D. Batory. Evolving Object-Oriented Architectures with Refactorings. To appear in *ASE '99*.

Permanent Address: 46-237 Heeia Street
Kaneohe, Hawaii 96744

This dissertation was typed by the author.