On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns *

Adam C. Jensen & Betty H.C. Cheng
Department of Computer Science & Engineering
Michigan State University
East Lansing, MI, USA
{acj,chengb}@cse.msu.edu

ABSTRACT

Maintaining an object-oriented design for a piece of software is a difficult, time-consuming task. Prior approaches to automated design refactoring have focused on making small, iterative changes to a given software design. However, such approaches do not take advantage of composition of design changes, thus limiting the richness of the refactoring strategies that they can generate. In order to address this problem, this paper introduces an approach that supports composition of design changes and makes the introduction of design patterns a primary goal of the refactoring process. The proposed approach uses genetic programming and software engineering metrics to identify the most suitable set of refactorings to apply to a software design. We illustrate the efficacy of this approach by applying it to a large set of published models, as well as a real-world case study.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Distribution, Maintenance & Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Design

Keywords

search-based software engineering, object-oriented design, refactoring, design patterns, software metrics, evolutionary computation, intelligent search

*This work has been supported in part by NSF grants CCF-0541131, IIP-0700329, CCF-0750787, CCF-0820220, CNS-0854931, Army Research Office grant W911NF-08-1-0495, Ford Motor Company, and a Quality Fund Program grant from Michigan State University.

GECCO'10, July 7–11, 2010, Portland, Oregon, USA.

1. INTRODUCTION

Maintaining an object-oriented design for a piece of software is a difficult, time-consuming task. Designing that software to be easily maintained and extended in order to satisfy new requirements is even more difficult, as it forces developers to consider not only the details of the solution space (e.g., how to build on-board software for automobiles) but also details of the problem space (e.g., how to manage a large and ever-changing number of AutoPart subclasses and their object instances). These difficulties led to the development of two important enabling technologies in software engineering. First, *metrics* enable a software engineer to evaluate various characteristics of design quality, such as flexibility or readability, in a mathematically rigorous fashion [1]. Second, *design patterns* provide a context-driven solution template for solving design problems that occur frequently in large software projects [6]. However, using metrics and design patterns often requires significant intellectual investment, thus limiting their adoption.

This paper addresses these concerns by introducing an approach, based on genetic programming, that automates the use of software engineering metrics to generate refactoring strategies that introduce design patterns. The approach is modular, thus enabling the reconfiguration or substitution of metrics and design patterns as appropriate. In addition, the output from the approach includes the specific set of refactoring steps to perform in order to apply the strategy that was generated. This information facilitates either manual or automatic application of the refactoring strategy, and we defer this decision to the software engineer.

Prior approaches to automated design refactoring have focused on making small, iterative changes to a given software design. For example, such an approach might identify so-called "god classes" that have a large number of public fields and methods. A god class can be split into a number of smaller classes, and its fields and members can be distributed among them. If the design is being evaluated by a metrics suite that analyzes class size, then such a change reflects favorably on the quality of the design. Ó Cinnéide [9], as well as Sunye *et al.*[14], introduced sets of refactorings that, under well-specified conditions (e.g., when OCL preconditions are satisfied), can be automatically applied in order to restructure a software design. However, the rule-based nature of these refactorings limits their general applicability. In contrast, evolutionary computation (EC) harnesses the exploration and exploitation capabilities of Darwinian evolution to find the best solutions to an optimization problem, including many software engineering problems [3, 4, 10, 11,

13]. While these approaches have demonstrated that evolutionary computation can be harnessed to generate novel sequences of design refactorings, their use of simple, iterative refactorings is not expressive enough to effect complex design changes. Specifically, they do not support the composition and interaction of multiple design changes in order to construct design artifacts such as design patterns.

In order to address this problem, this paper introduces REMODEL, an evolutionary computation-based approach to automatically generate design refactorings. REMODEL supports composition of design refactorings and makes the introduction of design patterns a primary goal of the refactoring process. This shift in focus carries three key benefits. First, we leverage the generally accepted notion that judicious application of design patterns improves the reusability and maintainability of non-trivial software designs [12]. Second, evolutionary computation enables us to explore the effects of composite refactorings to generate more innovative refactoring solutions that incorporate design patterns. Third, the approach illustrates the use of a modular framework for search-based refactoring whose components (i.e., metrics, design patterns, and design change mechanisms) can be easily substituted as appropriate for a given software project. By automatically generating refactoring strategies that are highly rated by software engineering metrics and also introduce design patterns, we reduce the cognitive labor for software engineers by suggesting design changes that are tailored to the software design in question. Furthermore, REMODEL provides the sequence of refactoring steps through which the input design can be transformed into a design that implements the suggested refactoring strategy, thus making it amenable for manual application by developers as well as inclusion in automated design tools.

We use genetic programming (GP) to identify the most suitable set of refactorings to improve software design. As with any evolutionary approach, GP comprises three components: a solution representation, a mechanism for making changes to that solution, and finally a means of measuring the solution's quality. In this approach, a solution is a refactored software design *and* the set of steps to transform the original design into the refactored design. Therefore, our solution representation comprises two key elements: first, a *design graph* that represents the software design that is being refactored; and second, a *transformation tree* whose nodes represent small, well-defined changes to the software design that is being refactored. These changes are known as *minitransformations*, leveraging work by Ó Cinnéide [9]. The minitransformations, when composed, are capable of creating instances of design patterns. The minitransformations make specific changes to the software design and are defined as functions that accept inputs and return outputs. As such, we developed an approach to express them concisely as nodes in a transformation tree, thus making them a strong candidate for use in a GP solution. These nodes gather input from their child nodes, which in turn represent elements of the software design that is being analyzed. In order to evaluate the quality of refactored designs, we use the QMOOD [1] metric suite that combines a rich set of metrics for analyzing the complexity of object-oriented hierarchies, cohesion of classes, and so on. QMOOD has also been used in prior search-based refactoring work [11].

We illustrate the efficacy of REMODEL by applying it to a large set of published models [7], as well as a real-world case study [5]. The remainder of the paper is organized as follows. Section 2 gives background information related to our approach. In Section 3, we present a detailed discussion of REMODEL. Section 4 contains a discussion of the experimental validation that we performed. Finally, we present our conclusions and future work in Section 5.

2. BACKGROUND AND RELATED WORK

This section reviews background concepts that support REMODEL, including metrics for object-oriented designs, design patterns, and related work.

2.1 Metrics

As the word suggests, a *metric* is a mechanism for measuring a specific aspect of an element, with a specific focus on how that aspect changes over time. In software engineering, metrics are used to measure characteristics of software such as lines of code, number of classes, cohesion among classes, and so on. They provide both an instantaneous snapshot of these characteristics and, when applied over time, a profile of how a software design has changed through its lifetime.

The hierarchical Quality Model for Object-Oriented Design (QMOOD) introduced by Bansiya and Davis [1] comprises 11 individual metrics, each of which evaluates a distinct aspect of object-oriented design quality. Notably, these metrics are designed to be amenable to automated evaluation, thus making them ideal for use in software engineering tools. From these metrics, Bansiya and Davis derived a set of formulas that measure abstract quality characteristics such as extensibility, readability, and maintainability. Such high-level quality characteristics can inform software developers of the ways in which their software is well-designed, and conversely, where it can be improved. Additionally, the QMOOD authors recommend that a real-valued weight should be assigned to each quality characteristic in order to specify its relative importance, thus providing quantitative feedback regarding how well a software design meets its design quality goals.

Two levels of validation were performed to evaluate the QMOOD quality model. The individual quality characteristics were evaluated first, followed by QMOOD's overall ability to estimate software quality. The authors chose the Windows Foundation Classes and the Borland Object Windows Library (OWL) as the design suite on which to validate the individual characteristics. They showed that the results from QMOOD agreed with the expected change in quality characteristics over time. To validate QMOOD's overall quality estimation, a group of 13 independent evaluators with experience in commercial software development were asked to develop a set of design and implementation heuristics that influence software development. The evaluators then assigned each heuristic to one or more of the QMOOD quality characteristics and scored 14 independently-developed projects according to how well each project satisfied the heuristics. The authors demonstrated a statistically significant positive correlation between the QMOOD-determined quality assessment of the projects and the 13 evaluators' assessments, thus demonstrating that the QMOOD quality model can effectively measure design quality.

2.2 Design Patterns

One popular type of structural improvement that is applied to software designs is the application of a *design pat-*

tern. Design patterns are small, reusable solutions to common design problems that occur in a specific context [6]. In this paper, we focus on a subset of the so-called Gamma design patterns, including `Abstract Factory`, `Adapter`, `Bridge`, `Decorator`, `Prototype`, and `Proxy` [6]. This subset of the Gamma design patterns addresses problems related to classes and their associations that make up the structure of an object-oriented software design. We refer to the process of applying a design pattern to a software design as design pattern *instantiation*. This term is used in software engineering when discussing the instantiation of an object from a class, and we extend its meaning to include the realization of a design pattern in a concrete software design.

2.3 Related Work

Extending automated refactoring to include the introduction of design patterns, Ó Cinnéide [9] proposed a series of refactorings called *minitransformations* that, when composed, interact to create instances of design patterns. The minitransformations create indirection or increased abstraction between classes by loosening coupling between those classes. For example, when applied to a software design in which `Maker` class creates many instances of a `Product` class, two `PartialAbstraction` minitransformations can interact to create an instance of the `Abstract Factory` design pattern, which facilitates construction of future subclasses of both `Maker` and `Product`. The minitransformations enable software developers to incorporate design patterns into their software design with minimal manual refactoring labor. However, design patterns contain multiple *roles* that must be filled by classes. For instance, the `Automobile` class discussed previously fills the role of *Factory* because it needs to produce many object instances. Thus, the approach proposed by Ó Cinnéide requires human input in order to decide the set of classes that will fill the roles of a given pattern.

Search-Based Refactoring.

O'Keeffe and Ó Cinnéide [10, 11] propose a technique in which an evolutionary algorithm decides the optimal set of refactorings to apply. They represent the source code and object-oriented structure of the program being refactored using an abstract syntax tree. To evaluate the quality of a refactored program, they use a fitness function based on the QMOOD metrics suite. Seng *et al.*[13] proposed a similar approach that used a genetic algorithm to optimize the class hierarchy of an object-oriented program by moving methods from one class to another and evaluating the modified designs using software engineering metrics. While these approaches are promising, they do not consider the introduction of design patterns as a key mechanism for refactoring software designs.

3. REMODEL: REFACTORING DESIGNS

This section introduces REMODEL, our approach to refactoring object-oriented software designs. REMODEL comprises a genetic programming environment (hereafter, "a GP") that is guided by software engineering metrics to determine the optimal set of refactorings to apply to a software design. This approach has two objectives: first, to improve the quality of the design as measured by the QMOOD suite of object-oriented metrics; and second, to introduce design patterns when appropriate in order to improve the maintainability of the target software design. In this section, we describe components of REMODEL, including how individuals in the evolving population are represented, how those individuals are changed through mutation, and how newly-evolved individuals are evaluated.

3.1 Individual Representation

In general, a GP contains a finite population of individuals that are given an optimization problem to solve. In REMODEL, these individuals are represented as a pair that includes a design graph and a transformation tree. A visual depiction of this arrangement is given in Figure 1.

3.1.1 Design Graph

The *design graph* represents the software design that is being refactored. For simplicity, we refer to this design as the *target software design*. The target software design is based on a UML class diagram, an example of which is given at the bottom of Figure 1, into a graph structure that is readily analyzed and manipulated. The `Driver`, `Convertible`, and `ConvertibleFactory` classes in the UML class diagram are translated into vertices (ovals) in the graph, and the relationships among those classes (labeled `drives` and `produces` in the UML class diagram) are represented by labeled edges between those vertices. In order to provide the GP with richer design information, we augment the design graph with semantic details such as class instantiations and function calls. When the GP population is initialized, each individual is given its own copy of the design graph to modify.

3.1.2 Transformation Tree

The second component of an individual, its *transformation tree*, is an encoding of a set of changes to the individual's design graph. When the tree is executed, it performs these changes and produces a modified (refactored) version of the design graph. Executing the tree involves visiting each of its nodes, beginning with the root, and executing that node's children recursively from left to right. Each node is either a *transformation* node that performs modifications to the design graph, an *information* node that informs and supports the modifications that are made by a transformation node, or a root node that acts as a placeholder. Each transformation node is an implementation of one of Ó Cinnéide's minitransformations [9]. Since the minitransformations require input in the form of classes, interfaces, and operations to perform their work, we need a mechanism to provide this input. Information nodes fill this role by representing elements from the design graph. The information nodes are attached as child nodes of the transformation nodes, and when the tree is executed, information flows from the leaf (information) nodes upward in the tree to the transformation nodes and eventually the root node. In Figure 1, `PartialAbstraction` is a transformation node, and `Driver`, `Convertible`, and `ConvertibleFactory` are information nodes.

This individual representation is novel and carries the following key benefits. First, our use of a graph to represent the target software design enables efficient computation of metrics that can be implemented as graph algorithms, such as QMOOD. Second, the order of the nodes in the transformation tree describe a step-by-step series of refactorings that can be applied to the target software design, thus supporting both manual and automated design refactoring.

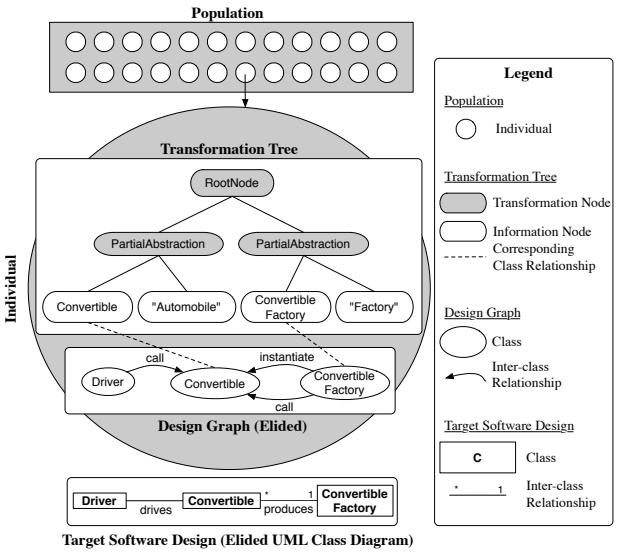

Figure 1: Relationships Among GP Elements

3.2 Design Change Mechanisms

In REMODEL, an individual's transformation tree is responsible for making changes to its design graph. Six distinct types of transformation nodes can exist in a transformation tree. These transformation nodes are an implementation of Ó Cinnéide's six minitransformations. We now describe each transformation node type in turn.

3.2.1 Transformation Nodes

Abstraction.

The `Abstraction` transformation node constructs a new interface that contains all of the public methods in an existing class, thus enabling other classes to take a more abstract view of the original class and any future classes that implement the interface. Introducing an interface in this fashion is a common refactoring step that restructures a software design to support additional functionality. For example, software that tracks vehicle inventory for an automotive dealership needs to track multiple vehicle types. These differing vehicle types will have many common characteristics, and a corresponding software design will include a class inheritance hierarchy that reflects those characteristics. When given a representative prototype, such as a class representing a mid-sized car, the `Abstraction` minitransformation can construct an interface for all classes that represent vehicles. A new vehicle-related class, then, only needs to implement the interface in order to integrate with the existing design.

Abstract Access.

The `Abstract Access` transformation node modifies a class `Context` that directly uses another class `Concrete` so that `Context` accesses `Concrete` through an interface `IConcrete`. This change is illustrated in Figure 2. It is generally accepted that such *decoupling*, or loosening of the relationships between two classes by using an interface, facilitates the development of new classes and other design changes in order to meet future requirements [6]. This minitransformation, when used in combination with the `Abstraction` minitransformation that creates a new interface from an existing class, uses that newly-created interface in order to decouple one class from another.

Delegation.

The `Delegation` transformation node is used to move part of an existing class to a component class, and to set up a delegation relationship from the existing class to its component. A class that has accumulated too many methods (sometimes called a "god class") may benefit from moving some of those methods to a separate *component* class. Since a method that is moved from one class to another will no longer have access to private members of the original class, those members must be made public. Thus, `Delegation` can require the public interface of a class to change.

EncapsulateConstruction.

The `EncapsulateConstruction` transformation node weakens the binding between a class that creates instances of another class by relocating the code statements that perform instantiation into a dedicated method. In general, a class that contains many code statements that create objects is difficult to maintain. For example, if there is a change in the parameters of a class's constructor, then all statements that instantiate that class are invalidated and must be updated to use the modified parameters. However, we can resolve the problem by moving all object-creating code statements for a class `Shape` to a dedicated method `CreateShape` that is responsible for constructing and returning instances of `Shape`. By creating this dedicated method, we guarantee that any change to the constructor of class `Shape` only affects the `CreateShape` method, thus reducing refactoring effort. The `EncapsulateConstruction` minitransformation constructs the dedicated method and modifies the relevant code statements to use the new method.

PartialAbstraction.

The `PartialAbstraction` transformation node constructs a new abstract class from an existing (concrete) class and adds an `inherits` relationship from the concrete class to the abstract class. Growing the class inheritance hierarchy in a software design is a common task as the design matures and as requirements change over time. `PartialAbstraction` grows the inheritance hierarchy by creating a new abstract class that has the same methods as an existing class, thus enabling other classes to inherit the functionality of the existing class in a way that facilitates future maintenance.

Wrapper.

The `Wrapper` transformation node wraps the functionality of an existing class with another class. Requests (e.g., method calls) to a wrapper object are forwarded to the wrapped object, and similarly the responses to those requests are passed back to the wrapper class and returned to the original calling object. This enables the wrapped class to be replaced at run time and loosens the coupling between the wrapped class and classes that use it.

3.2.2 Mutation Operators

To explore the space of refactoring strategies, this approach uses subtree crossover and point mutation in the transformation tree. Due to the nature of the nodes in the tree, a strict typing system is used to ensure that newly generated transformation trees are syntactically valid. After being selected, 90% of individuals undergo both subtree

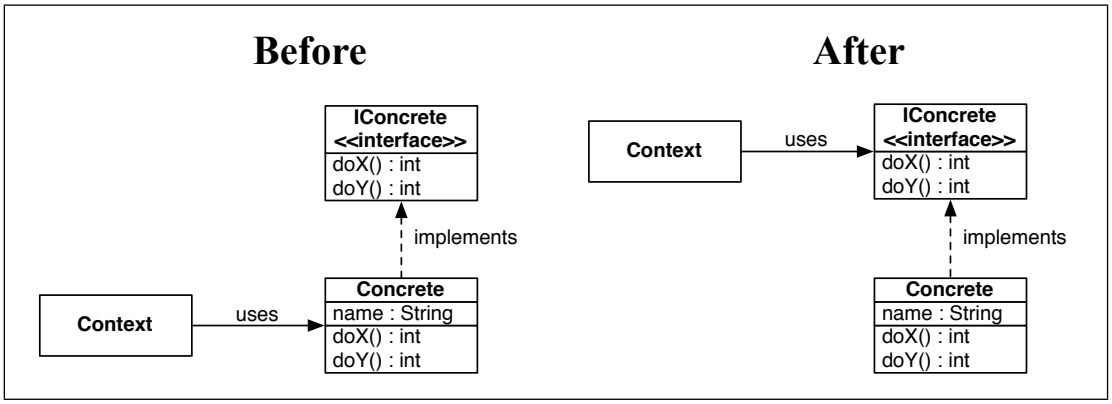

Figure 2: `Abstract Access` **Minitransformation**

crossover and mutation. The remaining 10% of individuals are reproduced as-is into the population that forms the next generation. During subtree crossover, non-leaf nodes are selected 90% of the time and leaf nodes are selected the remaining 10% of the time.

3.2.3 Example

We now present an example that shows the execution of a transformation tree and highlights the changes that it makes to a software design (represented by the design graph). A graphical depiction of the transformation tree and the UML representation of the software design are shown in Figure 3a and Figure 3b, respectively.

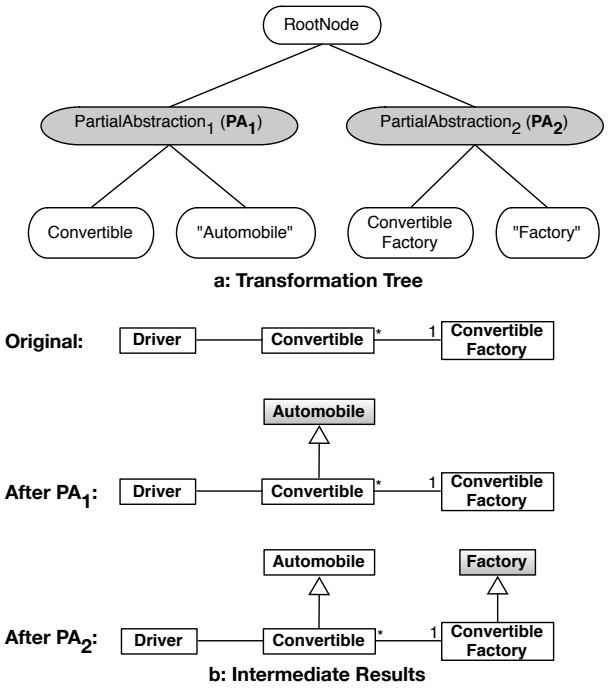

Figure 3: Example Transformation Tree Execution

This example illustrates the construction of an instance of the `Abstract Factory` design pattern. The transformation tree is executed using a post-order traversal. The three UML class diagram fragments represent, beginning with the top-most fragment, the state of the software design at the start of the tree traversal, after $\texttt{PartialAbstraction}_1$ is executed, and finally after $\texttt{PartialAbstraction}_2$ has been executed, as denoted by the annotations on the left side of the diagrams. The shaded ovals in Figure 3a are minitransformation nodes, and the shaded rectangles in Figure 3b denote the classes that are added at each respective execution step. In the original design, there are three classes: `Driver`, `Convertible`, and `ConvertibleFactory`. While this design is effective when only one type of vehicle (a convertible) needs to be considered, it quickly breaks down when new vehicles are added. To address this situation, we can modify the design to be more generic by introducing an abstract `Automobile` class and modifying the existing `Convertible` class to inherit `Automobile`. The next node to be executed is $\texttt{PartialAbstraction}_1$, and it is responsible for introducing the `Automobile` class. Its two child (information) nodes inform the $\texttt{PartialAbstraction}_1$ node of which class to use as the concrete class (`Convertible`), as well as the name to give the new abstract class ("Automobile"). The first modified software design, denoted by $\texttt{After PA}_1$, reflects this change.

Now that we have introduced the abstract `Automobile` class, we observe that the `ConvertibleFactory` class is only capable of making instances of the `Convertible` class, and there may be other subclasses of `Automobile` that need to be instantiated. To solve this problem, we can construct another abstract class called `Factory`. The $\texttt{PartialAbstraction}_2$ node makes this change, and its child information nodes specify that `ConvertibleFactory` should be the concrete class that inherits the new abstract class named "Factory". At this point, there is a functional[1] instance of the `Abstract Factory` design pattern in the software design. The final software design is denoted by the label $\texttt{After PA}_2$. Finally, `RootNode` is evaluated, and the execution of the transformation tree is complete.

3.3 New Design Evaluation

Next, we turn our attention to the way in which the quality of refactored designs is evaluated. After a population of individuals is constructed, each individual undergoes a

[1]We note that the `Driver` class should now communicate with the new `Automobile` class, but this change is outside the scope of the example.

fitness evaluation. During this evaluation, the individual's transformation tree is executed and makes a series of changes to that individual's design graph. The fitness value that is computed for the individual is a function of the overall quality of the design graph. Design metrics and the presence of one or more design pattern instances are the two key elements that are used to define the fitness functions. The remaining fitness function elements capture two optimization strategies for design refactoring. In this section, each element of the fitness evaluation is described in turn.

3.3.1 Metrics

To assess the overall quality of a software design, we leverage Bansiya and Davis's [1] QMOOD metrics suite. QMOOD comprises 11 individual metrics, each of which evaluates a distinct characteristic of object-oriented design quality, such as cohesion (i.e., how focused the responsibilities of a given class are) and coupling (i.e., the degree to which a class depends on other classes). Notably, the QMOOD metrics are designed to be amenable to automated evaluation, thus making them ideal for use in software engineering tools.

3.3.2 Design Patterns

The design change mechanisms (i.e., minitransformations) are designed to create instances of the Gamma design patterns [6] when they are composed. However, in order to identify software designs that include design patterns we must have a mechanism for detecting design pattern presence. In this section, we describe the Prolog-based design pattern detector that we developed for this work.

Prolog [8] is a declarative logic programming language that supports queries over relations. To facilitate design pattern detection, we translate the target software design into relations that specify the classes and operations in the design as well as the relationships between them (e.g., class inheritance or class-operation ownership). The relations for classes, interfaces, and operations are `cls`, `interface`, and `operation`, respectively. With this set of relations in hand, we construct one Prolog query for each design pattern whose instances we wish to detect in a given software design. The Prolog query for the Abstract Factory design pattern is shown in Figure 4. The queries in our implementation are adapted from QL queries given by Birkner [2].

```
abstract_factory(AFact,CFact,AProd,CProd,Client) :-
  cls(AFact), cls(CFact), cls(AProd), cls(CProd),
  cls(Client), inherit(CFact,AFact),
  inherit(CProd,AProd), instantiate(CFact,CProd),
  fcall(Client,CProd), AFact \= CFact,
  AProd \= CProd, AFact \= Client,
  CFact \= Client, AProd \= Client,
  CProd \= Client, AFact \= AProd,
  AProd \= CFact.
```

Figure 4: Prolog code for Abstract Factory design pattern

When an individual in the population is being evaluated, the Prolog design pattern detector analyzes the individual's target software design for the presence of design pattern instances. When an instance is detected, Prolog returns the set of design elements that satisfy the query; i.e., the set of classes, interfaces, and operations that participate in the design pattern instance. In the case of the Abstract Factory design pattern, this set of elements must include classes that fill the following roles: *Abstract Factory*, *Concrete Factory*, *Abstract Product*, *Concrete Product*, and *Client*. REMODEL records this set of design elements for later processing. (As an aside, we also used the Structured Query Language [SQL] to detect design patterns, and SQL returned results significantly faster than Prolog. However, we presented the equivalent Prolog queries here for simplicity of understanding. It should be noted that we were able to swap the Prolog and SQL detection engines in a straightforward fashion, as an illustration the modularity of REMODEL.)

3.3.3 Optimizations

While the QMOOD quality and presence of design pattern instances provide a sound overall measurement of design quality, we have identified two optimizations to steer the evolutionary process toward useful refactoring strategies.

The first optimization encourages individuals to evolve transformation trees with fewer transformation nodes. The number of transformation nodes in a transformation tree roughly corresponds to the number of refactorings that a developer must perform in order to realize the design changes suggested by this approach. Therefore, encouraging the evolution of fewer transformation nodes means that the burden on the developer will be smaller and the suggested design changes will be easier to understand.

The second optimization rewards individuals that incorporate sequences of transformation nodes that are known to produce design pattern fragments in green-field scenarios [9]. By using these sequences as templates, we believe that individuals will be able to discover similar, yet novel, refactoring strategies more efficiently than they would if they had to discover the sequences independently.

3.3.4 Fitness Function

The fitness function is given by the following formula:

$$F = Q + C_{PR} \cdot PR - C_{NCP} \cdot NCP + C_{MSR} \cdot MSR,$$

where Q is the quality of the individual's design graph as measured by the QMOOD metrics suite, PR is the amount of fitness bonus given for the presence of at least one design pattern instance, NCP is a fitness penalty proportionate to the number of transformation nodes in the individual's transformation tree, and MSR is a fitness bonus given for the presence of specific sequences of transformation nodes. The last three terms have a real-valued coefficient (e.g., C_{PR}) that determines the relative weight of that term. The optimal values for these coefficients is an open question, and we present empirical results from experiments using several sets of values in the next section.

4. VALIDATION

In this section, we present the results of four empirical experiments and analyze the results of a case study of a large, Web-based software system known as REMODD. The empirical experiments apply REMODEL to a set of 58 published software designs [7], each comprising 5 to 15 classes. The first experiment provides a baseline assessment of the approach, using only the QMOOD metric suite to determine fitness. The remaining experiments determine optimal values for the coefficients in the fitness function. Each experiment considers several values for a single coefficient, and

once the best value is determined, it is used for the remaining experiments. The resulting fitness function is used to apply REMODEL in a large case study.

4.1 Environment and Implementation

These experiments were conducted on a large cluster of PCs running SuSE Linux Enterprise Server 10. Our prototype GP was constructed using the Evolutionary Computation for Java (ECJ) framework.[2] The Java package JGraphT[3] provided support for graph analysis, and JLog[4] provided Prolog support.

Each of the five experiments comprised 100 generations. Tournament selection was used with a tournament size of 7. When tallying results, we considered only the most-fit individual at the end of each run. REMODEL was applied to each design run five times, using a unique random seed for each run, in order to acquire an unbiased sample.

4.2 Results and Discussion

Exp. I: Baseline.

The first experiment provides a baseline assessment of REMODEL, using only the QMOOD metrics suite to measure individual fitness. A total of 3,103 new design pattern instances evolved in the 290 runs (58 software designs and five runs per design) for an average of 10.70 instances per design. These initial results demonstrate that REMODEL is capable of introducing design patterns in a diverse set of software designs.

Exp. II: Penalizing Large Tree Size.

The second experiment evaluates a mechanism for penalizing individuals in proportion to the number of transformation nodes in their transformation trees. This number is approximately equal to the number of refactoring steps that the tree represents, and our goal is to apply a small pressure toward a smaller number of refactoring steps in order to ensure that a software engineer can understand the refactoring strategies that REMODEL generates. We used the values 0.0025, 0.025, and 0.25 as the coefficients on the NCP term. The value 0.025 yielded the best set of results with an average of 5.86 transformation nodes, and we used this value for the C_{NCP} coefficient in the remaining experiments.

Exp. III: Rewarding Design Pattern Instances.

The third experiment evaluates a mechanism for rewarding individuals whose refactoring sequences introduce at least one new design pattern into the target software design. We used the values 0.125, 0.25, 0.5, 1.0, and 2.0 as the set of candidate coefficients for the PR term. After measuring the percentage of the 58 software designs that showed a statistically significant ($p < 0.05$) increase in the number of design patterns when each value was tested, we observed that the value 1.0 yielded the greatest percentage of designs. Therefore, we used 1.0 as the value for the C_{PR} coefficient in the remaining experiments.

Exp. IV: Rewarding Transformation Sequences.

The fourth experiment evaluates a mechanism for rewarding individuals whose refactoring sequences contain subsequences that are known from work by Ó Cinnéide to construct design pattern fragments [9]. After measuring the percentage of the 58 software designs that showed a statistically significant ($p < 0.05$) increase in the number of design patterns when each value was tested, we observed that the value 1.0 yielded the greatest percentage of designs. Therefore, we used 1.0 as the value for the C_{MSR} coefficient in the remaining experiments.

[2] `http://cs.gmu.edu/~eclab/projects/ecj`
[3] `http://jgrapht.sourceforge.net`
[4] `http://jlogic.sourceforge.net`

4.3 Case Study: ReMoDD

Next, we present the results of a case study in which we applied REMODEL to a large, Web-based software system known as the Repository for Model-Driven Development, or REMODD [5]. REMODD contains model-driven engineering artifacts, including models, test cases, documentation, and code. The REMODD design comprises 23 classes and includes instances of design patterns, so this experiment also evaluates REMODEL's ability to construct instances of design patterns in the presence of existing instances. In this experiment, we used the coefficient values that were selected in the previous four experiments. Specifically, $C_{PR} = 1.0$, $C_{NCP} = 0.025$, and $C_{MSR} = 1.0$. We conducted five runs for this experiment, and each run was assigned a unique seed in the range [1,5] to introduce random variation and avoid bias.

4.3.1 Results

The results of the case study are summarized in Table 1, which shows the number of new instances of each design pattern that evolved. The numbers shown are the totals from the five independent runs. The average number of transformation nodes in the best individuals in each run was 4.50.

Pattern	# of Instances
Abstract Factory	2
Adapter	13
Bridge	3
Composite	0
Decorator	2
Prototype	26
Proxy	15

Table 1: Evolved Pattern Instances in Case Study

A sample design pattern instance that evolved in the REMODD design is shown in Figure 5. The classes that participate in the pattern instance are enclosed in the shaded rectangle. Three application classes are shown to illustrate how the instance attaches to the rest of the design. The evolved design pattern is an instance of `Abstract Factory` and therefore includes the following roles: Abstract Factory, Concrete Factory, Abstract Product, and Concrete Product. These roles are filled by the classes `User`, `Viewer`, `AbstractNewClass`, and `Review`, respectively. The class `AbstractNewClass` was created during the execution of a `PartialAbstraction` transformation node.

4.4 Discussion

This case study demonstrates that REMODEL can be applied to a large software design. An average of 12 new design pattern instances evolved in the best individuals, thus providing plenty of suggestions for a software engineer who

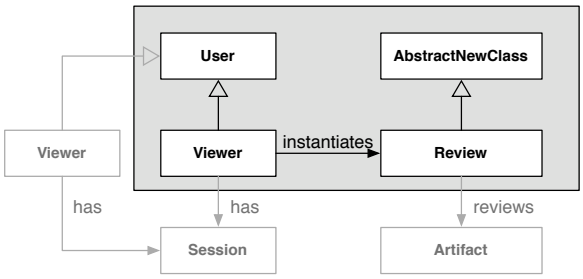

Figure 5: Evolved Design Pattern Instance

might be considering the introduction of design patterns to make the REMODD design more extensible or maintainable. However, this broad set of suggestions also highlights the need for human review and demonstrates that this automated approach should not be blindly followed.

We note that the arbitrary class name `AbstractNewClass` is a placeholder that should be replaced. The transformation nodes are not equipped with natural language processing mechanisms, and therefore it is difficult for them to choose sensible names for new classes. One possible solution to this difficulty is to choose names from a dictionary of nouns taken from a requirements document. However, this solution may still produce class names that do not make sense to a developer. In order to preserve the full automation of REMODEL and minimize confusion, we recommend that the choice of new class names be deferred to a human developer when an evolved pattern instance is being applied.

5. CONCLUSIONS

In this paper, we presented REMODEL, an approach for automated refactoring of software designs that combines genetic programming, software engineering metrics, and minitransformations to introduce design patterns in existing software designs. Specifically, we showed that the minitransformations support composition of multiple design changes, thus enabling the generation of richer refactoring strategies than were previously possible. We conducted a set of four experiments to determine the optimal coefficients for the terms in our fitness function. Finally, we applied REMODEL to a large, Web-based software system. Our results show that the approach is capable of simultaneously improving the quality of a software design with respect to metrics as well as automatically introducing design pattern instances, a combination that was not previously considered. Our ongoing research considers the use of different metrics, domain-specific design patterns, and design change mechanisms with different levels of abstraction to explore the impact of these factors, as well as the modularity of REMODEL, on different refactoring problems.

6. REFERENCES

[1] J. Bansiya and CG Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[2] Marcel Birkner. Object-Oriented Design Pattern Detection using Static and Dynamic Analysis in Java Software. Master's thesis, University of Applied Sciences Bonn-Rhein-Sieg, Sankt Augustin, Germany, 2007.

[3] M. Bowman, LC Briand, and Y. Labiche. Multi-objective genetic algorithm to support class responsibility assignment. In *IEEE International Conference on Software Maintenance, 2007. ICSM 2007*, pages 124–133, 2007.

[4] J. Clarke, JJ Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, et al. Reformulating software engineering as a search problem. *IEEE Proceedings - Software*, 150(3):161–175, 2003.

[5] R. France, J. Bieman, and B.H.C. Cheng. Repository for model driven development (REMODD). *Lecture Notes in Computer Science*, 4364:311, 2007.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Reading, MA, 1995.

[7] P. Gomes, F.C. Pereira, P. Paiva, N. Seco, P. Carreiro, J.L. Ferreira, and C. Bento. Using CBR for Automation of Software Design Patterns. *Lecture Notes in Computer Science*, pages 534–548, 2002.

[8] U. Nilsson and J. Małuszyński. *Logic, programming and Prolog*. Wiley, 1990.

[9] M. Ó Cinnéide. *Automated Application of Design Patterns: A Refactoring Approach*. PhD thesis, University of Dublin, Trinity College, 2001.

[10] M. O'Keeffe and M.Ó. Cinnéide. A stochastic approach to automated design improvement. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 59–62. Computer Science Press, Inc. New York, NY, USA, 2003.

[11] M. O'Keeffe and M. Ó Cinnéide. Search-based refactoring: an empirical study. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(5), 2008.

[12] L. Prechelt, B. Unger, W.F. Tichy, P. Brössler, and L.G. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, pages 1134–1144, 2001.

[13] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1909–1916. ACM New York, NY, USA, 2006.

[14] G. Sunye, D. Pollet, Y. Le Traon, and J.M. Jezequel. Refactoring UML Models. *Lecture Notes in Computer Science*, pages 134–148, 2001.