2009

Automated Transforms of Software Models: A Design Pattern Approach

Brandon Adam Gump
Wright State University

AUTOMATED TRANSFORMS OF SOFTWARE MODELS:

A DESIGN PATTERN APPROACH

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

By

BRANDON ADAM GUMP
B.S., Wright State University, 2008

2009
Wright State University

WRIGHT STATE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

November 23, 2009

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Brandon Adam Gump ENTITLED Automated Transforms of Software Models: A Design Pattern Approach BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science.

Thomas C. Hartrum, Ph.D.
Thesis Co-Director

Mateen M. Rizki, Ph.D.
Thesis Co-Director

Thomas A. Sudkamp, Ph.D.
Department Chair

Committee on
Final Examination

Thomas C. Hartrum, Ph.D.

Mateen M. Rizki, Ph.D.

Travis E. Doom, Ph.D.

Joseph F. Thomas, Jr., Ph.D.
Dean, School of Graduate Studies

ABSTRACT

Gump, Brandon Adam. M.S., Department of Computer Science, Wright State University, 2009.
Automated Transforms of Software Models: A Design Pattern Approach.

In the realm of software development, projects are plagued by continuous maintenance at the source code level as well as tedious transformations from formal specifications to source code. Such work consumes a large amount of time only to create complicated, un-intelligible, and un-reusable code that is completely detached from initial design rational. To cope with these problems, The Air Force Institute of Technology (AFIT) Wide Spectrum Object Modeling Environment (AWSOME) was designed to generate specifications that can be transformed into abstract designs and finally into source code. The specifications are written in the AFIT Wide-spectrum Language (AWL) and parsed in by the tool into a meta-model.

The focus of this thesis is to expand AWSOME's transform capabilities by automating the application of design patterns to existing ASTs by altering their structure. Automating the application of design patterns to existing software models offers many advantages including extending reusability and easing maintenance.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

Table Page

ACKNOWLEDGEMENT

This thesis would not have been possible without the help of the following persons: Dr. Hartrum, for his dedication to aiding me in every aspect of this undertaking, Drs. Doom and Rizki, for participating on my final examination committee, previous AWSOME students, for the work they contributed to the tool in the past, and my friends and family, for supporting me through all the work I have done.

DEDICATION

This thesis is dedicated to my loving mother, Deborah Gump. Throughout my academic career, she has always been there to encourage me to do my best. Her reassurance has been invaluable to forcing me to commit to my work and finish in times I felt completely overwhelmed. Thank you, for being a wonderful mother.

I. INTRODUCTION

It is no secret that large scale software systems are not created error free in their entirety from a single pass of a software lifecycle. In fact, it is the case that most projects' costs (as much as 70%) are expended in maintenance of code [1]. Many of these maintenance tasks revolve not around bug fixing, but rather adding new functionality to existing software. Classical models, such as the waterfall model, which require each phase (e.g. design, implementation, etc.) to be completed before beginning the next phase, make additive software enhancements difficult. To combat this setback, the idea of evolutionary development has surfaced in which a software model is created from initial specifications. Such a model can be iteratively improved and reused as new specifications and requirements emerge [2]. The focus of this thesis is to design transformations that can automatically be applied to existing software models while supporting the idea of software evolution and preserving original system behavior.

1.1 AWSOME

In order to define a system capable of taking an existing software model that can be iteratively improved via transforms, it is helpful to break the process into pieces. Typically a transformation system begins with domain knowledge stored in a model created by an engineer. Using the domain model, an engineer then constructs a formal specification during the problem setting phase of development. The system then uses a series of transformations and an engineer's input to alter the requirements specification into a design specification. Once the design specification is obtained, more transforms can be applied by the system to create executable source code for a target platform. The

entire process and its participants are outlined in Figure 1.1. The underlying idea of such an approach is that while the system can automate parts of the process, the tool should be thought of as an aide to a software engineer who is responsible for deciding which transformations the system should apply [3].

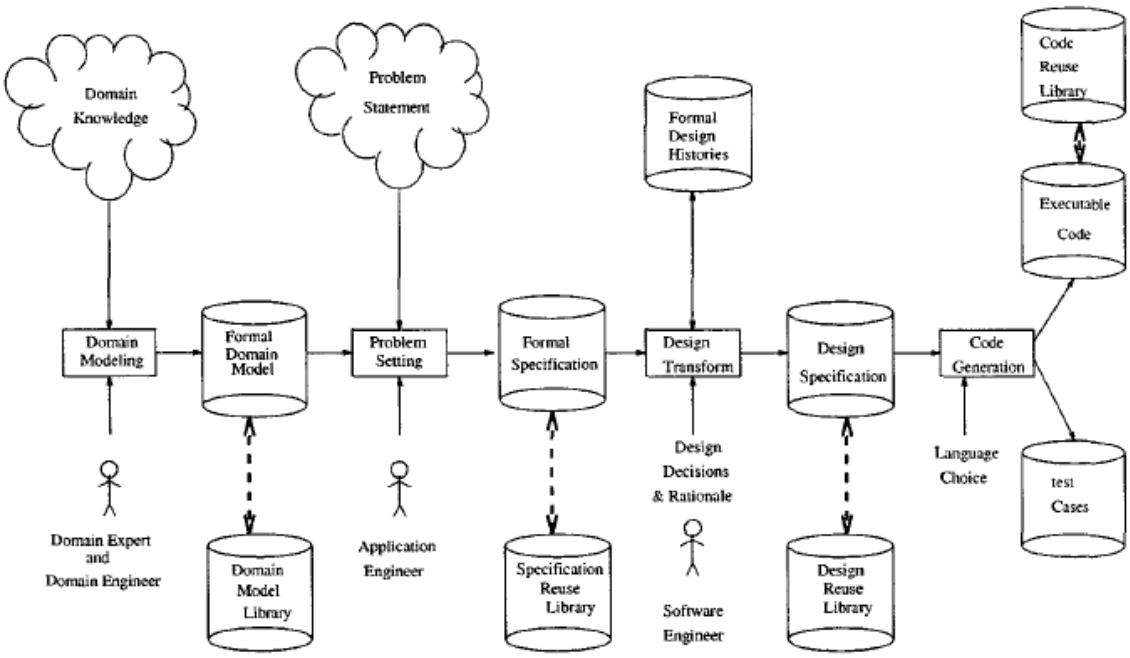

Figure 1.1: A typical transformation system [3].

The Airforce Institute of Technology's Wide Spectrum Modeling Environment (AWSOME) is a realization of such a transformation system. The system begins by parsing in a formal specification written in AWSOME's Wide-Spectrum Language (AWL). AWL is based on the Object Constraint Language (OCL) and as such employs syntax that makes it possible to design models of software systems that include classes, associations, finite-state dynamic models, as well as class invariants and pre/post-conditions. As shown in Figure 1.2, the language is very similar to that of a standard object-oriented programming language [4].

```
package testpackage is
   type CHAR is abstract;
   type STRING is sequence of CHAR;

   class Person is
       private name : STRING;

       public procedure setName (theName : in STRING)
        assumes True
        guarantees name' = theName
        is begin
             name := theName;
          end;

       public function getName () : STRING is
          begin
             getName := name;
          end;
   end class;
end package;
```

Figure 1.2: Example of AWL syntax.

Once the domain model has been parsed in, it is stored in the system as an abstract syntax tree (AST). An AST can be thought of as a model that represents the entire software system as a collection of object classes. Each object class is a structural model containing the class's attributes, methods, and dynamic model (to represent state transitions). While the system outlined in Figure 1.1 implies a sequential approach that first creates a domain model, then a formal specification, and then a design specification, the AWSOME tool is structured to allow the three steps to be mixed together and interactively applied after the initial model is parsed in by the tool. Once the user is satisfied with the transformations that have been applied to the original model, a final transformation can be applied to the AST that turns the model into Java source code [3].

1.2 Design Patterns

Throughout a programmer's career, it is often the case that the notion of programming déjà vu will surface: the feeling that the problem he or she is tackling has already been solved. Problems such as accessing an aggregate of objects sequentially without

exposing implementation details or providing an interface to simplify interaction with a complicated system are but a few examples that come to mind. From a programmer's standpoint, it would be convenient if there were documented design strategies that proposed solutions to these common problems and produced re-usable, flexible software. This is precisely the goal that design patterns address through four components [5]:

1. Each design pattern has an associated **name** that serves as an identifier to quickly elucidate the problem, solution, and consequences related to the design pattern without detailing low level specifics.
2. Certainly a design pattern would be of little utility if it had no purpose. For this reason, design patterns also describe the **problems** they are intended to solve. In some cases, the problem description of a pattern can serve as a checklist of necessary conditions that warrant the use of the pattern.
3. The heart of each design pattern is the **solution** itself. Since the goal of patterns is to produce reusable and flexible code, the solution does not describe a concrete implementation in a particular language. Rather, the solution describes an abstract approach to organizing software components, such as classes and functions, to solve problems.
4. While it may seem that a design pattern should be applied if it is an appropriate solution to a problem, the **consequences** of applying a pattern should be evaluated before applying it. These trade-offs are typically related to time and space, but could directly affect the system's extensibility and reusability. Furthermore, the pattern may propose variants in the solution that each has its own advantages and disadvantages.

1.3 Problem Statement

Currently the AWSOME tool provides support for several transformations such as automatically generating *get* and *set* methods for all attributes specified in a model and transforming various post-conditions to logical statements. While such transformations are useful because they automate part of the tedious, error-prone process of generating object-oriented code, transformations that alter the entire model structure (such as introducing entire new classes) would be a useful addition to AWSOME's library.

In developing this thesis, we hypothesize that it should be possible to create AWSOME transforms that would allow an engineer to introduce design patterns into an existing meta-model. Furthermore, in testing this hypothesis, we intend to develop a methodology that would allow additional design pattern transforms to be developed beyond the ones presented in this thesis. Testing this hypothesis involves altering existing structural and functional components of AWSOME models to directly support various design patterns. This thesis implements transformations for each of several design patterns that may be directly applied to a model in order to alter its structure to implement the functionality provided by each pattern. Since some patterns may require input from a user before they can be applied, the AWSOME tool will be updated to query such information from the user before applying such a transform. In addition, the transformations include functionality to determine whether or not a pattern can actually be applied to an existing model.

1.4 Appling Patterns

Since design patterns do not specify code level implementations, automated application seems appropriate at a higher level of abstraction. For this reason, the

patterns will be applied at AWSOME's meta-model level via transformations that alter the structural model's (AST) classes' internal attributes and functions (the dynamic model will remain unchanged). We postpone taking an AWL file through a complete design pattern application until Chapter 2.

1.5 Thesis Scope

Clearly, there are far too many design patterns to explore in this thesis alone. Furthermore, there are some patterns that are impossible to pursue due to limitations underlying AWSOME itself (for example, some patterns rely on multiple inheritance, which is not possible in the tool). Since restructuring the entire system is not viable, the thesis will instead focus on patterns the existing architecture can support. However, minor changes necessary to support patterns will be implemented to make the tool more versatile. The patterns to be considered (though not necessarily implemented) include abstract factory, adapter, builder, composite, decorator, façade, factory method, flyweight, iterator, observer, singleton, strategy, and visitor. For each pattern, a detailed description will be presented as well as a specification to describe what is to be expected when the transform is applied. Furthermore, the underlying design and implementation of each transform will be discussed. In order to test the transforms contributed by this thesis, we design and anchor an interactive GUI, *GumpPanel*, to the existing AWSOME tool as well as create several input AWL test files. The discussion of the testing material is postponed until Chapter 5.

1.6 Approach

We note that creating transforms in AWSOME itself to modify an existing AST is not a revolutionary idea contributed by this thesis. Some examples include Venkata's work

with transforms that made it possible to generate executable Java code as well as Swamy's work to transform associations in a formal specification [6, 7]. In fact, we follow the typical strategy of previous work by consolidating each of our transforms into subclasses of a special class in AWSOME named *Transform*.

1.7 Outline

This thesis addresses the issue of implementing the automation of applying design pattern transforms to existing AWSOME ASTs in an object oriented approach. As such the remainder of the document is divided into the following chapters. Chapter 2 develops a rich background of AWSOME as it relates to the thesis as well as details regarding the design patterns included as transforms. Chapter 3 is structured as a requirements analysis where the feasibility of implementing each pattern is explored and expected results of each design pattern transform are clearly defined. Furthermore, the concepts of when patterns are both applicable and appropriate are explored in this chapter. Chapter 4 addresses the specification of the previous chapter through an in-depth design of the implementation of each transform. Chapter 5 illustrates the test results of implemented transforms. The chapter focuses on showing application of the transforms on existing example models. Chapter 6 closes with conclusions drawn from this thesis as well as suggestions for future work.

II. BACKGROUND

2.1 AWSOME

2.1.1 Abstract Syntax Tree (AST)

If AWSOME could be thought to have a heart, truly the abstract syntax tree (AST) used to internally represent a meta-model of a system would be it. The AST directly stands between the initial input specification and the final output of the system. The AST itself will be the target of modification as various transformations are applied to the system. As such, the idea of how the meta-model represents a particular system must be explored.

At the highest level, an AST is merely a collection of object classes that represent the system being modeled. In essence, an object class can be thought of as an entity that contains the name of the class, attributes related to inheritance hierarchies, as well as the following three components:

- A structural model which consists of all the attributes defined in the class. Some design pattern transformations require inserting new attributes into a class. The model supports the addition and removal of attributes to an existing object class, though the functional model may have to be altered to support changes in attributes.
- A functional model which consists of all the methods defined in the class. Again, there are design patterns that will require adding or removing functions. The model also supports the addition and removal of methods to existing object classes, so applying patterns transformations may also alter the

functional model.

- A dynamic model which consists of states and transitions that define the class's behavior. Since design patterns alter the internal structure of the model in order to support additional functionality and reuse, the notion of a class's transition through some state space is unnecessary. As such, the thesis will not alter any dynamical models that may be associated with an object class.

2.1.2 AFIT Wide Spectrum Object Modeling Environment (AWSOME)

Recall that the typical approach taken by a semi-automated transformation system is that of a sequential approach. In that regard, the system begins with a formal domain model encompassing knowledge. The model is then altered by an engineer to create a formal specification (essentially requirements) based on the problems the system needs to solve. Finally, correctness-preserving transformations can be applied to the formal specification to generate a design specification. The design specification itself can be thought of as a model of the system that can be directly transformed to source code for the target platform.

While initially AWSOME supported this step-by-step process for transformations, the system has been updated to support a more interactive approach. AWSOME now begins with a formal specification defined in an AWL file that is parsed in to form the initial AST. Once the AST has been created, the tools incorporated in AWSOME are free to modify the model in any of the development stages. This means that an engineer is free to apply domain knowledge or formal specifications to a model as desired as well as utilize design transformations that modify the AST. Once the user is satisfied, he or she has the options to generate Java source code from the AST and/or generate a new AWL

file that could be used later again as a formal specification. Figure 2.1 illustrates the access provided to the AST by AWSOME.

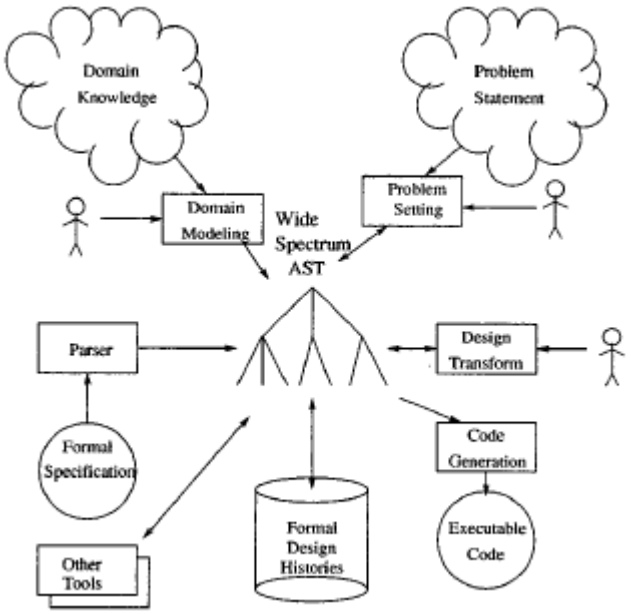

Figure 2.1: AWSOME Transformation System [3].

While the way in which AWSOME allows interaction with the AST is clear, the notions of domain model, formal specification, and design specification must be elaborated as they apply to the system. A domain model may be thought of as a description of a system's relationship between the entities it is composed of. As an example, consider a simple library system with two main entities: libraries and books. In such a system, the relationship in the model would be that of libraries having books. The results of modeling this system in AWL can be seen in Figure 2.2.

```
package librarysystem is
type char is abstract;
type String is sequence of char;
   type integer is range * .. *;
type BookArray is array[integer] of Book;

class Library is
      private books : BookArray;
   end class;

class Book is
      private title : String;
end class;

end package;
```

Figure 2.2: A domain model of a simple library system.

A formal specification builds on the model by specifying what, not necessarily how, the system's components should do. The description is typically presented in mathematical notation; in the case of AWSOME, specifications that closely mimic the object constraint language (OCL) are written in AWL as a series of pre and post-conditions for methods in each class. The results of adding formal specifications for the constructors of components of the library system can be seen in Figure 2.3.

```
package librarysystem is
   type char is abstract;
   type String is sequence of char;
   type integer is range * .. *;
   type BookArray is array[integer] of Book;

   class Library is
     private books : BookArray;
     public class function Library (Books : in BookArray) : Library
      assumes True
      guarantees this.books' = Books and Library = new Library
      is begin end;
  end class;

   class Book is
      private title : String;
      public class function Book (Title : in String) : Book
     assumes True
     guarantees this.title' = Title and Book = new Book
     is begin end;
   end class;
end package;
```

Figure 2.3: A formal specification for constructors in a simple library system.

The final component, the design specification, is the direct product of applying transformations to the formal specification. The transforms work to take the pre and post-conditions and turn them into statements inside of the methods they account for. In the case of the library system, our post-conditions simply turn into assignment statements as can be seen in the constructor methods in Figure 2.4.

```
package librarysystem is
   type char is abstract;
   type String is sequence of char;
   type integer is range * .. *;
   type BookArray is array[integer] of Book;

   class Library is
      private books : BookArray;

   public class function Library (Books : in BookArray) : Library
      is begin
          this.books := Books;
          Library := new Library;
     end;

   end class;

   class Book is
      private title : String;

     public class function Book (Title : in String) : Book
         is begin
        this.title := Title;
        Book := new Book;
     end;

   end class;

end package;
```

Figure 2.4: The result of transforming the specifications into statements.

With the idea of these three components developed, it should be clear how the model allows the user to add new domain model knowledge and specifications to the AST and allow the system to transform the additions. When the user is satisfied with the changes made to the AST, he or she can then generate executable Java code or a new AWL description of the system model. An example of Java code generated can be seen in Figure 2.5. We note that the AWSOME type *BookArray* appears as a class in Java

instead of an array. Transformation of such data types into Java code is beyond the scope of this thesis.

```
package librarysystem;
/**
* File    Book.java
*/

public class Book {
   protected    String title;

   public  Book(String Title) {
       this.title = Title;
   }
}

package librarysystem;
/**
* File    Library.java
*/
public class Library {
   protected    BookArray books;


   public  Library(BookArray Books) {
        this.books = Books;
   }
}
```

Figure 2.5: The result of transforming the AWL in Figure 2.4 to Java code.

2.2 Design Patterns

At their heart, design patterns provide descriptions of common problems in computer science as well as potential design solutions to said problems. As such, many common programming techniques that could loosely be deemed design patterns, such as data encapsulation or object oriented techniques, have surfaced. In the following section, we attempt to outline many of the commonly accepted design patterns originally appearing in the "Gang of Four" [5].

2.2.1 Singleton

In some software packages, it is the case that a particular object class may be required to only ever have one instance. Possible situations where this may arise include

centralized accumulator objects such as a logger and an object to be shared among classes. The ability to capture this functionality is directly provided by the singleton pattern.

The singleton pattern is used to ensure that a class has only one instance and to provide a global point of access to the instance. The pattern achieves this goal by making its constructor private to prevent outside instantiation. To allow a user access to an instance, a private static attribute of the class type is created and instantiated within the class itself. The instantiation is performed in a static class method that instantiates the object only once and stores it in the private static member of the class. Furthermore, said function provides access to the object by returning a reference to it to its caller [8 pp359-369].

2.2.2 Abstract Factory

Consider a software package with a user interface that supports several different "looks" or "themes." Clearly each "theme" would share notions of generic components such as buttons and windows, but each "theme" would have its own version of such objects. With a large degree of variability in component types, it would be desirable to provide a common interface that allows a client to create related objects together. Furthermore, it would be beneficial to provide access to these components without forcing the user to specify their concrete classes. The ability to encapsulate related types of object creators is precisely the functionality provided by the abstract factory pattern.

The abstract factory is applied through the use of an abstract creator class responsible for declaring an interface for each possible component the client could need produced. For each component theme, a subclass is created to implement the creator interface which

ensures that all created objects will share the same "look-and-feel." In addition, abstract classes for each component the creator could make are also used. For each of these abstract classes, a concrete subclass of that component is created for each "theme." With this system in place, the client is free to interact with the abstract creator in order to obtain component instances. Since the client does not know the concrete subclasses of components, it interacts with them via a defined interface which allows the client to plug in any version of a component without knowing the concrete implementations [5 pp87-90].

2.2.3 Factory Method

It is often the case that subclasses in a hierarchy of classes inherit methods from their parents. As such, the subclasses often override functionality provided by the parent class implementation. For example, at some times an application may need to log to a file and other times it may log to the console, clearly much of the functionality of logging is common in either case. This leads to applications being forced to pick particular subclasses based on the functionality the program requires at the time the choice is made. Clearly this would mean that a particular application would have to be aware of all existing subclasses as well as provide potentially complicated logic to decide which subclass should be used at a particular time.

The factory method pattern serves to remedy these problems by providing a function capable of using application context and other factors to instantiate an appropriate subclass and return it as an instance of the parent class type. The method is implemented by providing an abstract class that declares the factory method and having concrete subclasses that supply a concrete implementation of the method. Such a method is

declared to return the super class type, and provides logic to decide which subclass object to instantiate and return [9 pp 65-71].

2.2.4 Adapter

In developing reusable classes, it is a primary goal that the classes provide various services that a variety of clients would find useful. Sometimes, classes may provide utilities a client would like to make use of, but the interfaces provided by the classes are not what the client expects. For example, we may have a client that requires validating taxes for a U.S. customer through a specific method such as *validateTaxes* that we must now update to support validating Canadian taxes. In this case, we may have an existing class that accomplishes this, but the method to do so may be *validateCTaxes*. Rather than revamp the interface of the other class, we can provide an adapter class that provides the *validateTaxes* method a client expects, but it turns around and makes use of the *validateCTaxes* function.

In order to achieve this goal of providing an interface the client expects, the adapter pattern offers two strategies. The first strategy is that of an *object adapter* in which the adapter class inherits the interface expected by the client while holding an instance of the adaptee class. Calls by the client to the adapter are transformed into appropriate calls to the adaptee class. The other technique is that of a *class adapter* in which the adapter inherits both the interface the client expects as well as the interface provided by the adaptee class. In this case the interface methods are overridden to make calls to the adaptee class's methods [5 pp139-145].

2.2.5 Builder

Typically object construction involves initializing appropriate class attributes, a

practice usually done in the constructor. When object initialization is simple, such a technique suffices, but in some cases the underlying construction of an object may be complicated and could be accomplished in several different ways. For example, we may have a class designed to represent a maze, but in order to actually build mazes we have to create a number of walls, doors, and rooms. Clearly this would create a very complicated maze class constructor and changing maze types would require remaking the entire class. Using this builder pattern, the complicated work of creating a huge maze is moved out of the super class and instead subclasses of the maze class can be plugged in to build different kinds of mazes.

The builder pattern accomplishes this by providing an abstract builder class that specifies methods that are responsible for creating one part of the entire desired product. Sub classes of the builder class are responsible for concrete implementations that build a product in a specific way as well as provide a way for the end product to be returned. In addition, another class is created to direct the builder's construction of products [5 pp97-105].

2.2.6 Composite

For any type of object in an application, it can either be considered a single component or it may be a collection of other components (including single components or more collections). A novel example of such a case is that of a file system, in which folders may hold other folders or files. Although folders and files are different, clearly a program could be simplified if a common interface to access either with operations such as *open* was created. The very goal of the composite pattern is to provide a common interface to treat individual components and collections in the same way.

The composite achieves this behavior by first defining a super class responsible for declaring the interface required for all objects in the composition. Subclasses of this class are created to represent individual objects and implement the desired behavior for the interface methods. The pattern also creates a subclass to represent the composite of parts which can hold more composites or individual objects. As such, the class has methods to add or remove components as well as implementation of the required super class methods as they relate to collections of components [9 pp137-139].

2.2.7 Decorator

Typically we add functionality to existing classes by changing their original source or sub classing them and adding new methods to the subclass. However, sometimes we would prefer to add functionality to an object dynamically in situations where we cannot do the former or when inheritance may lead to an explosion of subclasses. For example, we may have a text field in our editor that we want to put a plain scroll bar around and then we later decide to put a fancy border around. In addition, it may be the case we would later want to take some of these features away, which would not be possible with inheritance. The decorator pattern aims to make it possible to add or remove such features to an existing object in this dynamic fashion.

The decorator pattern accomplishes this functionality via an inheritance hierarchy that at the top level is comprised of a class that defines an interface for objects that can have functionality added to them. Concrete subclasses of this class can define actual components to which functionality can be attached as well as their behavior. Another abstract subclass of this parent class referred to as the decorator contains a reference to a component and defines an interface that the component requires. Concrete subclasses of

the decorator class are responsible for actually adding additional functionality to the component [5 pp175-176].

2.2.8 Façade

Sometimes it is necessary for a client to interact with several different classes in order to accomplish a certain task. Consider for example a compiler system: the program must be parsed in, then compiled, then linked with appropriate libraries, then built into an executable. Clearly making the client interface with each of these components directly would create a very complicated system. The façade pattern attempts to simplify this process by providing functionality the client needs through a simple interface that masks all the work necessary to actually fulfill a request.

The façade achieves this functionality by creating a class that the user can interface directly with instead of the complicated subsystems. As such, the façade class provides methods that accomplish specific tasks the client could want. Inside of these methods, the complicated calls to the various classes necessary to complete the request are implemented. All the user need do is call the appropriate function and the façade handles all the details of completing the request [5 pp185-189].

2.2.9 Flyweight

It is often the case that applications use a large number of objects. For example, consider a text processing program where each character is an object. Clearly this would be costly in terms of memory if it were necessary to allocate objects for every single character in a very large document. A much more useful method would be to store the information intrinsic (the same) to each character in a single object that can be used whenever the application needs an instance of that object. The flyweight pattern is meant

to allow clients to share as much information common to objects as possible while still allowing extrinsic information about objects to be maintained.

The flyweight achieves this goal via a factory class responsible for creating flyweight objects and managing the sharing of them among clients. In addition, the flyweight pattern uses an abstract flyweight class that species an interface through which flyweight objects can be acted on. Subclasses of the abstract flyweight class typically include classes that make use of intrinsic state information (and store it within the class) and classes that allow use of extrinsic state information [5 pp195-200].

2.2.10 Iterator

It is often the case that a program may contain a container of some sort comprised of objects or primitives. Furthermore, we may want to give clients a method to retrieve elements without exposing the underlying implementation in case it later must change. It is exactly the goal of the iterator pattern to provide various ways for a user to iterate through a list without directly exposing the list implementation.

The iterator design pattern is accomplished through the use of two abstract classes, one of which specifies the interface to create an actual interator object and another which specifies the interface iterators must provide (operations such as *next* and *first*). Concrete subclasses of these classes are then created to provide the client with different types of iterators to navigate collections of objects in various ways (such as alphabetically) [5 pp257-258].

2.2.11 Memento

In some applications it is often the case that it would be desirable to provide the user with a means to undo operations committed. In order to do so, it is necessary to store all

relevant data about an object at various checkpoints so that internal state can be restored. This storage is precisely the intent of the memento pattern: a memento object can be created to encapsulate relevant internal state information for some object at some point in a program's execution. Later, memento objects can be passed back to the object they hold the state of and allow the object itself to readjust its state without exposing underlying implementation details to the user.

In order to support this functionality, the memento pattern makes use of a separate class that contains internal attributes that can store all information that represents the state of the class an ability to undo is desired for. As such, the new memento class must have support for setting and getting its internal state in order to provide functionality to store or restore old state as necessary. In order for a class to be able to support this pattern, it must be updated with a method that takes a memento object and restores its previous state from it as well as a method that creates a memento object using the current state of a class object. Once both of these classes have been updated appropriately, another class is created (or it may already exist) with the intent of storing the memento objects and being responsible for passing mementos to the class they hold state for [5 pp283-285].

2.2.12 Observer

Often it is the case that a system may be divided into several classes that work together to provide a program's required functionality. In such a case, it is very desirable that said objects maintain a consistent state and notify each other class regarding relevant changes in state. The observer pattern aims to provide a simple solution for objects interested in the change of state of another object to be automatically notified when the other object changes state.

The observer pattern achieves this goal through the use of a subject class that allows observers to be attached or detached, and as such keeps track of all observers. Concrete subclasses of said subject class store the state of interest to observers and send notifications to observers when it changes. As such, an observer class also exists that provides an interface for updating objects that should be notified of subject changes. Concrete observer subclasses may also store the state to stay consistent with the subject as well as contain a reference to the subject [9 pp331-335].

2.2.13 Strategy

Sometimes it is the case that we have several different algorithms for accomplishing a task. For example, we may have different techniques for encrypting data, and we want to be able to vary the techniques on the fly. Clearly it is undesirable to build all of the strategies into one class filled with conditional logic to decide which algorithm is appropriate. The strategy pattern allows a family of algorithms to exist by encapsulating each into a class that can be used interchangeably by a client.

In order to achieve this functionality, the strategy pattern employs an abstract strategy class that specifies an algorithm interface. Concrete sub classes contain specific implementations of algorithms to implement required functionality. The actual class that uses the strategy class maintains a reference to one of the concrete subclasses which it uses to do the work [5 pp.315-319].

2.2.14 Visitor

Although classes are created with as much functionality as the creator can foresee a need for, it is often the case we would like to add new operations to existing classes. For example, a class may print mathematical expressions in prefix order, but we would like to

also be able to print them in in-order and postfix notations. Clearly it is undesirable to change the original class to include the new functionality as that would require recompiling the entire class. It is precisely the intent of the visitor pattern to allow a programmer to easily implement new functionality into existing classes without modifying the original classes.

The visitor pattern accomplishes this by creating a visitor class hierarchy that defines *visit* methods for concrete sub classes to implement that accept instances of concrete classes we want to define new operations on. In order to pass these *visit* functions instances of concrete subclasses, the original object classes must be updated to support an *accept* method that takes the visitor class as an argument and simply turns around and calls the visitor's *visit* method, passing itself as an argument [9 pp179-185].

2.3 Related Work

While the notion of other work involving transforms that can be applied to a software model is noteworthy, we are interested in previous work regarding automation as it applies to design patterns.

We note that there are varying strategies of automation that have surfaced in this field as described by Bulka. The first strategy, templates, is achieved by tools that allow a designer choose a pattern to generate a customized template. The drawback of such a method lies in the fact that a user must manually tailor the generated code to meet application needs. The second strategy, parameterized templates, includes tools that allow a user to specify existing classes to adapt a chosen pattern to as well as customize aspects of the pattern itself before generating code. The author notes that the advantage of these tools is that little manual editing is necessary after generating code, though the

need to specify classes, methods, and attributes before generating code causes a heavy amount of user interaction. Another strategy, which the author refers to as "super" parameterized templates, follows the footsteps of the previous strategy, adding a larger amount of customization to the pattern implementation itself. The allure of such a method is in its ability to allow the designer to more readily fit a pattern to his needs. We provide a description of one such tool later in this section. The final strategy, dynamic intelligent patterns, takes the automation one step further, noting which classes and methods participate in the pattern and updating them automatically as the system changes. While more complicated than other techniques, dynamism has the benefit of preserving pattern integrity. For example, if a visitor pattern was present in the model and the user adds a new class, the appropriate methods to visit the new class will be added to all existing visitor classes [10].

In his evaluation of several automation tools, Bulka notes that some patterns are more suitable to be automated than others. Furthermore, even with a large amount of customization for a pattern implementation, it is still possible that an automatically generated pattern may not exactly fit the designer's needs. On the other side, the automated pattern may contain far more than the designer needs, leading to bloated code that is difficult to understand. Truly, the prospect of automation is beneficial in that tool wizards can educate a designer about the pattern he is applying in terms of design trade-offs and implementation, but it may never be possible to fully automate patterns that require a large amount of customization [10].

As an example, we consider a "super" parameterized template tool, Cogent, designed by Budinsky, Finnie, Yu, and Vlissides. The tool created by this team is split

into two separate parts: an information repository and a code generation page. The information repository can be likened to a text book description of a pattern including design tradeoffs associated with its implementation. Each design pattern is given its own page that allows the user to specify the names of participants in the pattern as well as decide between any design tradeoffs. For example, if the user wanted to generate composite pattern declarations, he would choose the names of component, composite, and leaf classes. As far as trade-offs are concerned, one example in the composite pattern allows a user to choose whether or not to include child management operations in the base class [11].

Once the form has been completed, a user can use the tool to automatically generate C++ code for the pattern. It is the intent of this tool that the code then be added to an existing project that must be updated to utilize the pattern. At its heart, the tool transforms a user input into actual code. However, in this case the authors use three distinct participants that collaborate to achieve the goal of code generation. The first component specified is the "presenter" which is a frontend to communicate with the user to gather input as well as show the final code output. The third component used is a "code generator" which creates language specific code implementing a pattern. Sitting between the two is the second component, the "mapper", which functions to control how the "presenter" and "code generator" send information between each other to accomplish their tasks. In this decoupled design, the authors note that each piece of the system can be modified in isolation and the benefit of substitutable components surfaces. For example, one implementation uses a web browser as a "presenter" to gather input, a PERL interpreter that serves as a "mapper", and a "code generator" implementation

composed of several COGENT scripts [11].

While the area of automatically generating code for design patterns is relatively novel, work to automate the detection of design patterns present in existing code has also recently surfaced. Work in this area by Tsantalis, Chatzigeorgiou, Stephanides, and Halkidis have yielded a Java based system with the ability to identify many common design patterns in software systems with a large degree of success. The basic premise of their technique utilizes Java bytecode manipulation to expose the underlying static structure of a system. The information is then used to construct matrices that capture the ideas present in UML diagrams such as associations, generalizations, abstract classes, and method invocation. Graph matching algorithms are then used to detect similarities among the derived graphs and graphs that represent an implementation of the design pattern [12].

In related work, Gueheneuc and Antoniol created a system to detect design patterns present in code, but produced results that were only marginally successful in correctly identifying patterns present in large scale systems. In this case, the authors define a meta-model language, Pattern and Abstract-level Description Language (PADL), used to represent classes, fields, methods, interfaces, inheritance, and implementation relationships present in a system's source code. In creating models of code, a three level model is created. The first layer is the source code level model which identifies common elements such as classes, methods, inheritance, class relationships, and interactions. The second level is the idiom-level model which the authors note is used to expose binary class relationships related to composition, aggregation, and association. The third level is the design-level model used to expose micro architectures in the system using constraint

solving. Each level is compared with "motif" models which describe the generic form of a pattern. This comparison is used to identify matching patterns and class participants and interaction in the pattern [13].

As related work in the area of design pattern automation and detection has shown, the real secret lies in identifying class relationships and interaction. In essence, this is precisely the intent of design patterns: to specify classes and how they work together to accomplish a particular task. For this reason, it is obvious that any work to automate the creation of design patterns must work to create class hierarchies and interaction that fulfill the pattern requirements. Of course, it is clear that we can automate the process, but why would we want to? The work in this area holds promise in the realms of software maintenance and design. A great benefit can be gained if the implementation of a pattern can be automated to prevent errors that a human designer may introduce. Furthermore, the ability to automatically apply patterns and detect them greatly simplifies understanding of the inner workings of a system. Rather than take time to discover that a pattern exists in a system, a designer could use a tool to discover it and begin capitalizing on the design pattern rather than figuring out how it works. Truly, with utility in saving time and errors (read: reducing cost), the future for work in this area seems promising.

2.4 Summary

In this chapter, we surveyed many of the design patterns presented in the original design pattern book by Gamma [5]. In our descriptions, we gave an outline of each pattern as well as a brief description of a typical implementation of the pattern. The entire discussion of this chapter is meant to set the stage for Chapter 3, in which we use these descriptions to elucidate requirements the patterns will demand AWSOME to

satisfy in order to model them in our meta-model language, AWL. Furthermore, we use this requirements analysis to determine which patterns are feasible and/or worth implementing as transforms in AWSOME.

III. REQUIREMENTS ANALYSIS

3.1 Pattern Analysis

Before beginning our analysis of each pattern, we develop an example transform in 3.1.1 to illustrate what we consider in our analysis. Following this example are many analyses used to decide which patterns will be developed into transforms for this thesis. In doing so, pattern requirements that AWSOME must provide support for are specified and patterns are evaluated as to whether or not they may be implemented. While in this analysis we conclude that physical limitations of AWSOME do not restrict integrating most design pattern transforms, we see that the process cannot be fully automated. For this reason, the inputs required from a design engineer such that the transform can take the inputs and automate the process of applying the pattern are also explored. We use the amount of input required as well as the usefulness of providing a transform for a pattern to make a decision as to which patterns we will implement. The results of this exploration are summarized in Table 3.1 [5].

3.1.1 Example Pattern Application

In this section, we present an example detailing how and at which stage of AWSOME our design patterns will be applied. We postpone the discussion of the requirements a pattern imposes on AWSOME and our selection of which patterns to implement as transforms until the requirements analysis later in this Chapter.

As an example, assume we wish to write a class that will manage communication between a program and a database. As such, we would like to have only one connection to the database and force all communications to use this link. The initial (very

simplified) version of this class is described using AWL in Figure 3.1. Clearly as the class is defined now, there is no way to prevent the creation of multiple *DatabaseConn* objects as the constructor for the class is public. However, this problem seems like something that may come up often in software design and as such there should be a well defined solution already developed.

```
package testpackage is
   type char is abstract;
   type String is sequence of char;
   class DatabaseConn is
      private dbInfo : String;
      private conn : CONNECTION;
  public class function DatabaseConn () : DatabaseConn
   assumes true
   guarantees this.conn = new CONNECTION and DatabaseConn = new
   DatabaseConn is begin end;
... other relevant methods ...
... declaration of CONNECTION class omitted ...
end package;
```

Figure 3.1: Initial design of class to represent database connection.

Before we can continue with our transform, we require that the model be past the phase of specifying preconditions and post-conditions for methods. We impose this requirement because our transforms intend to create new classes, methods, and attributes as well as update statements in existing methods. To satisfy this need, we can apply existing transforms to the model in Figure 3.1 to turn the conditions into code statements. The result of altering the model with such transforms can be seen in Figure 3.2.

```
package testpackage is
   type char is abstract;
   type String is sequence of char;

   class DatabaseConn is
      private dbInfo : String;
      private conn : CONNECTION;

   public class function DatabaseConn () : DatabaseConn is
      begin
           this.conn'          :=          new          CONNECTION;
           DatabaseConn := new DatabaseConn;
      end;
   ... other relevant methods ...
   end                                                        class;
   ... declaration of CONNECTION class omitted ...
end package;
```

Figure 3.2: Design class modified to eliminate pre and post-conditions.

With the conditions transformed, we note that a design pattern exists to help us accomplish this goal of allowing only one instance of a class to be instantiated at a time. The pattern applicable in this case is the singleton, which allows only one instance of a class to exist and provides a global access point to the instance. Now that we have found an appropriate pattern that matches our requirements, we can apply the pattern's solution to our model. In the case of singleton, we must first convert the constructor to private to prevent outside instantiation of the class and introduce a private static attribute to the class that is of the same type as the class itself, call it *singleton*. Next we must introduce a method to provide a global point of access to the static attribute, call it *getInstance*. This method will be responsible for returning the instance of *singleton* to the function caller and initially instantiating the *singleton* the first time a caller requests it. The updated model after applying the specified transform can be seen in Figure 3.3.

```
package testpackage is
   type char is abstract;
   type String is sequence of char;

   class DatabaseConn is
      private dbInfo : String;
      private conn : CONNECTION;
      private singleton : DatabaseConn; //static, but can't show it
   private class function DatabaseConn () : DatabaseConn is
      begin
           this.conn           :=           new           CONNECTION;
           DatabaseConn := new DatabaseConn;
      end;

   public   class   function   getInstance   ()   :   DatabaseConn   is
      begin
          if ( this.singleton = null )
             singleton := new singleton;
          getInstance := singleton;
      end;
   ... other relevant methods ...
   end                                                          class;
   ... declaration of CONNECTION class omitted ...
end package;
```

Figure 3.3: Result of applying the singleton transformation to the DatabaseConn class

3.1.2 Singleton

In order to integrate the singleton design pattern into AWSOME, we require the ability to create static methods with the logic to instantiate and return static objects to the method caller. While AWSOME provides support for most of these requirements, we note that we will have to update AWSOME to support static attributes. The tasks required to add this functionality are detailed in 4.14. With these modifications, AWSOME can fully support the singleton pattern and thus we should be able to create a singleton transform.

To fully automate the application of the singleton to a model, we will require the AWSOME tool to retrieve some information from the user beforehand. In this case, we require that the user has specified which class is to be transformed into a singleton. Once

this task is complete we can automate the process of generating the necessary components of a singleton without further user input. We shall refer to such patterns as this that require little user input to automate as minimal patterns. Given the usefulness of the singleton pattern, the minimal amount of input required, and the functionality of AWSOME, we implement the singleton as a design pattern transform.

3.1.3 Abstract Factory

In order to implement the abstract factory pattern into AWSOME, we require that the system allow the creation of abstract classes and their concrete subclasses, as well as provide logic to select a particular theme of objects and allow instantiation of concrete products. AWSOME allows for all of these requirements to be satisfied and thus we should be able to add the pattern as a transform into the system.

The abstract factory again falls under the classification of a minimal pattern. In this case, we require that the user specify only abstract super classes for the concrete product classes. In addition, we force the user to specify the concrete product classes that fit into the same type (e.g. different "themes" of a button class). Once this information is acquired, the process of creating classes and functions to finish implementing the pattern can be fully automated. Given the usefulness of the abstract factory pattern, the minimal amount of input required, and the functionality of AWSOME, we implement the abstract factory as a design pattern transform.

3.1.4 Factory Method

The implementation of the factory method pattern into AWSOME requires the ability to create abstract classes, concrete classes that subclass them, and logic to decide which classes to instantiate and return. AWSOME is capable of achieving all these

requirements and thus we should be able to integrate the pattern into the system.

In this case we again note that the factory method is a minimal pattern. We require that the user specify abstract super classes that could potentially have concrete subclasses to be returned from the factory method. While we could make the user specify internal logic that stipulates which concrete class the factory method returns, we instead can opt for simple logic hardcoded in and force the user to conform to using said logic to pick which object is desired. Such an approach dramatically reduces the complication of the user specified input and makes the automation of apply the pattern possible. Given the usefulness of the factory method pattern, the minimal amount of input required, and the functionality of AWSOME, we implement the factory method as a design pattern transform.

3.1.5 Adapter

As previously noted, there are two different techniques for implementing the adapter pattern, and as such there are two different sets of requirements produced. In the case of a *class adapter*, we require either multiple inheritance or a way to implement interfaces and inherit from a class. Since AWSOME does not provide support for either, it not possible to implement *class adapters*. However, if we instead turn to *object adapters*, we require that AWSOME allow for the creation of classes that can subclass an existing interface, allow instantiation of an object of the adaptee type, and allow method calls to the adaptee class within the new adapter class. As AWSOME provides support for all of these requirements, the *object adapter* should be able to become a transformation in the system.

In evaluating the amount of user input required to automate the application of this

pattern, we find a need for substantially more input than the previous patterns. In this case, we would initially require an existing adaptee class to be specified by the user in some way. Furthermore, we must force the user to specify what methods the adapter class methods must call in the adaptee class in order to produce the required behavior for each of these functions. This essentially boils down to the user writing the functions himself through a graphical interface and the automation of the pattern mainly translating the user's input to the model. Due to the sheer amount of input required from a user, we refer to this type of pattern as a maximal pattern. Before implementing any design pattern, we must discover a way to balance the amount of input required with the amount of work the transform accomplishes in order for the transform to have any practical utility. In the case of the adapter, it seems difficult to reach such a balance. Although the adapter pattern is useful and existing functionality of AWSOME would allow us to implement *object adapters*, we choose not to implement a transform given the large amount of input required to produce minimal output.

3.1.6 Builder

In order to implement the builder pattern into AWSOME, we require the abilities to create abstract classes as well as concrete subclasses that implement the super class's required functionality, to instantiate pieces of an object through various functions, and to assemble the pieces into a complete product that can be returned to the requester. Currently AWSOME provides support for all of these needs and as such it should be possible to implement a builder design pattern transform.

In attempting to automate the builder transform, we see that it requires a significant amount of input. In this case, we require that the user specifies the abstract builder class,

but we can eliminate the need to specify concrete subclasses as we can search for them automatically in the model. In addition, we require the user to specify which methods (in order) are used to build parts of the object and which are used to get the final resultant product. In requiring the user to specify these, we see that the builder pattern is moderate pattern. Although AWSOME functionality allows the pattern to be implemented, we note that we are again essentially providing a GUI for a user to insert method calls into the AST and opt not to provide a transform for the builder design pattern.

3.1.7 Composite

There are several requirements that must be imposed on AWSOME in order to implement the composite transform. The system must be able to support the creation of new abstract classes with abstract methods as well as the creation of subclasses that implement the abstract methods. In addition we require that the system support a mechanism for storing collections of objects, such as an array. Since AWSOME supports these needs, it should be possible to create a composite transform in the system.

In order to automate the application of the composite pattern into an existing model, we require input from the user. To begin, the user must specify which classes will serve as leaf classes in the composite structure. In addition, the user must also specify which classes will serve as the composite classes. Furthermore, we require that user specify which methods from the leaf classes should be supported by the collection of objects. To cut down on the required input, we can use an array as the underlying container for storing objects within the composites. As this approach requires specifying methods and classes, we classify it as a more moderate pattern in terms of the amount of input required. While we note that the composite pattern has utility and that AWSOME

can support the pattern, creating a transform for this pattern would result in specifying a composite shell of sorts that sets up the class hierarchies and storage, but leaves out concrete methods. Due to time constraints, we do not create a transform for the composite pattern in this thesis.

3.1.8 Decorator

In order to implement a decorator transform into the existing AWSOME, we require that the following functionality is provided by the tool: the ability to create abstract super classes and subclasses that implement the required functionality of the super classes as well as the ability to store a reference to a class within a class. As the tool provides support for these simple requirements, it should be possible to implement a decorator transform into the tool.

The decorator design pattern is very specific in nature and would mandate that several of the classes required already exist, such as the decorator concrete classes and the concrete component. Assuming that they did exist already, the application of the pattern is simple and requires very little user input in the form of simply specifying the classes that fulfill the desired roles. Although this is all a user need supply, in reality a large amount of input is required as these classes must already exist in the model, which presumes the user already wrote classes that were providing nothing to the program. In this regard, the decorator can be seen as a maximal pattern. If however, the user was resorting solely to inheritance to add responsibilities to objects, it is possible for the user to specify the subclasses fulfilling decorator roles and transform them to adhere to the pattern. Assuming this, we may require additional input from the user to decide the contents of the component class. In this case, the decorator leans more toward a moderate

pattern. Although we note that the decorator design pattern can be implemented in AWSOME and is useful when inheritance is not viable, each implementation of a decorator is tailored specifically to the problem at hand, making it difficult to automatically generate anything. For this reason, we opt not to include a transform for the decorator design pattern.

3.1.9 Façade

In order to implement the façade pattern transform into AWSOME, we require the ability to be able to create new classes. Furthermore, we must be able to create methods in classes capable of accessing other classes and methods in the software system. As AWSOME supports the ability to accomplish both of these goals, we should be able to implement the façade transform into the system.

In the case of the façade, we quickly see that the required input to achieve the pattern is quite large. We require that the user identify which classes and methods (including arguments) calls must be utilized in order to achieve the functionality we wish to provide in the façade class. Given the fact that the user is using a GUI to essentially write code by specifying a multitude of method calls and arguments, the transform does very little in comparison as it simply translates the user's requests into updates of the AST. As a result, we classify the façade as a maximal pattern in terms of the input required. One potential way to combat this problem, however, is to allow the user to specify sequences of statements in the existing model that could be encapsulated into a single façade function to perform a specific task. Although the façade pattern has practical use and AWSOME can support its implementation, we opt not to create a transform for the pattern due to the large amount of input required as compared to the

relatively small amount of output produced.

3.1.10 Flyweight

There are a few requirements that AWSOME must provide support for in order to implement a flyweight design transform. The tool must allow abstract super classes to be created as well as concrete subclasses that implement the abstract classes' required methods, the ability to store references to objects created by the factory class, and a way to instantiate objects in a factory-like class and keep track of whether or not they have been created yet. Since AWSOME provides support for these operations, it should be possible to implement a flyweight transform into the tool.

Before the flyweight transformation can be applied, we require several pieces of input from the user. To begin, we require that the user identify all classes that will be transformed into flyweight objects. In addition, for each of these classes, it is necessary for the user to specify which attributes in the class are intrinsic, i.e. the same for each instance of the class. All other attributes can be inferred as extrinsic, i.e. variables specific for each instance of a class, and do not require user specification. Using this information the application of the pattern can then be automated. Since we require the user to specify many existing classes, methods, and attributes, we classify the flyweight as a moderate pattern. Although AWSOME's architecture provides support for the flyweight pattern's implementation, we note that most of the classes would already need to be present in the model and as such the transform would require the user to specify a large number of participants to essentially create a flyweight shell. For this reason, we opt not to include a transform for the flyweight design pattern in this thesis.

3.1.11 Iterator

In order to implement the iterator pattern in AWSOME, we require the following: a way to store a collection of objects such as an array, the ability to create abstract classes that outline an interface, and the ability to create concrete subclasses that implement super class methods capable of pulling elements out of a collection and returning them to the method caller. As AWSOME provides support for all these mechanisms, it should be possible to implement an iterator transform into the system.

The iterator pattern at a glance can require very little input from the user to automate the application of the pattern to the model. In essence, we can simply require the user to specify containers of objects in which the user would like to require an iterator to be used to retrieve the contained elements. However, if the container is not simply an array, we will require the user to provide methods that specify how to retrieve objects from the list and check the size of the list in order to implement the pattern. In addition, if we are to allow non-trivial methods of iterating through a list, such as alphabetically, the user must indicate methods necessary to put the underlying list in a particular order to support the iterator. Due to the amount of classes and methods that the user must specify for the iterator, we classify this as a maximal pattern. Although AWSOME would support an implementation of the iterator design pattern, we would not be able to automatically generate methods to iterate through the list other than in order and we would force the use of an array. Given these limitations, we opt not to include a transform for the iterator design pattern into AWSOME.

3.1.12 Memento

In order to implement the memento pattern into AWSOME, we require the ability to create classes and add them to the model. Furthermore, we require the ability to add

methods and attributes to existing classes as well as update variables in existing classes. AWSOME must also support the ability to call functions in other classes and pass objects to the functions. In typical implementations, the memento class is created as a private inner class inside of the class we wish to store the state of. Unfortunately, AWSOME does not support inner classes so we will resolve this issue by specifying the memento as a public class, acknowledging that this strategy violates the encapsulation issue. Since the tool currently supports all of these features, the memento pattern can be implemented into AWSOME.

While it may seem at first that all the input that is required for the memento pattern to be applied is the class for which we would like to create it, we argue that this may not be the case. In some situations, some attributes in a given class will not be relevant for restoring the state. In order to handle this, we may query the user for a class to create the memento for as well as what attributes from that class the user would like to store (consequently, we may save some user input by asking whether or not the memento should simply store all attributes). Once we have all of this information we can automate the application of the memento pattern. Since in some cases we require the user to specify all the attributes to store, we classify the memento as a moderate pattern in terms of input. In our approach, however, we avoid querying this additional information and instead choose to use all attributes in creation of the memento, causing the transform to require minimal input. Given the utility of the memento design pattern, AWSOME's ability to support an implementation, and the minimal amount of input, we include the memento transform in this thesis.

3.1.13 Observer

In order to implement the observer pattern as a transform in AWSOME, we require the ability to create abstract classes as well as concrete classes that implement their abstract methods, a way to store a collection of objects such as an array, and the ability to call methods from other classes within methods of other classes. As AWSOME provides this desired functionality, it should be possible to implement an observer transform into the tool.

At first glimpse, it appears as though the user would be required to specify many classes to serve as observers and concrete subjects to apply the observer pattern. However, this is not the case as we require the user to only specify which classes need serve as concrete subjects. Using this information, we can automate the creation of the abstract subject (or observable) class as well as all the related classes for the observer functionality. Since the input requires the user specifying classes only, we classify the observer design pattern as a minimal pattern. However, we note that classes typically take specific actions in their *update* methods, and as such we form an observer pattern shell in the model. Given the utility of the observer design pattern, AWSOME's ability to support an implementation, and the minimal amount of input, we include the observer transform in this thesis.

3.1.14 Strategy

Implementing a strategy transform into AWSOME requires the following functionality: a way to specify abstract super classes as well as concrete subclasses that implement desired functionality, the ability to store a reference to an abstract super class that may hold concrete subclasses, and the ability to call class methods on an object of an

existing class. As AWSOME provides methods to achieve all these requirements, it should be possible to implement the strategy pattern transform into the tool.

In the case of the strategy pattern, we again classify it as a minimal pattern. In this case, the user need only specify the classes that implement a particular strategy for an algorithm. If the classes do not contain the same method name for the function that actually executes the algorithm, we may require the user to further specify the appropriate method in each class. At any rate, this requires the user to specify only a few details before the pattern itself can be automatically integrated into the model. Despite the fact that AWSOME will support the strategy pattern, we note that in order to implement it there would likely already be a notion of separate algorithms to perform a particular task. As such, the usefulness of a transform for this pattern is questionable so we opt to not incorporate a transform for the pattern into AWSOME.

3.1.15 Visitor

In order to implement a visitor pattern transform into AWSOME, there are a few requirements we must impose on the system. We must be able to create abstract super classes that define interfaces implemented by concrete subclasses, call other class methods from within an existing class, and pass a reference to an object's self (i.e. a *this* identifier) as a method argument. Since AWSOME allows for all of these operations to be performed, we should be able to implement a visitor pattern into the system.

The visitor pattern also falls into the category of a minimal pattern. In this case, we require that the user need only specify the classes for which visitor support should be added. After this is known, the pattern application can be fully automated to introduce methods into the appropriate classes and create the related visitor classes. Given the

minimal amount of input, the extreme usefulness of the visitor design pattern, and the fact that AWSOME can support the pattern's requirements, we choose to include a visitor transform into the tool.

3.1.16 Summary

In this section, we outlined the functionality AWSOME would need to support in order to implement each design pattern. Furthermore, we also analyzed the input each transform would require in order to automate the application of the associated design pattern. Using both pieces of information and evaluating our time constraints, we concluded for which design patterns this thesis would construct transformations. The results of our conclusions are summarized in table 3-1.

Table 3.1: Summary of conclusions regarding design patterns discussed.

Pattern Name	Implementable	Input Complexity	Implemented in Thesis
abstract factory	yes	minimal	yes
object adapter	yes	maximal	no
class adapter	no	maximal	no
builder	yes	moderate	no
composite	yes	moderate	no
decorator	yes	maximal	no
façade	yes	maximal	no
factory method	yes	minimal	yes
flyweight	yes	moderate	no
iterator	yes	maximal	no
memento	yes	moderate	yes
observer	yes	minimal	yes
singleton	yes	minimal	yes
strategy	yes	minimal	no
visitor	yes	minimal	yes

3.2 Pattern Transforms

We note that our selection of patterns to implement from Table 3.1 is based on the following conclusion: while the amount of input required to automate a pattern varies, we

choose to implement patterns for which we can produce a larger amount of output as compared to input. For example, the façade requires a large amount of input and produces a large amount of output, but essentially the input is a specification of which methods to call and the output is an updated model that calls the specified methods. In our opinion, very little, if anything, is gained by producing a transform that automates the façade pattern. Due to time constraints, we do not attempt to discover ways to reconcile the input to output ratio in order to make implementing such patterns feasible. Also, time constrains prevent us from surveying every known design pattern (including those not listed in Table 3.1).

In this section, for each pattern that we implemented, we outline the situations in which we would use such a transform as well as the requirements that a model would have to satisfy before it is eligible for a given transform. Furthermore, the requirements that a transformed model must satisfy are also presented. In each section, we show a UML diagram that we designed using common knowledge of the design pattern being discussed to illustrate the goal of applying the pattern. In the case of some transforms, requirements relating to updating the rest of the model to use the transformed classes are discussed. We note that some patterns have different techniques for implementation. However, in order to produce requirements that we can create a design to satisfy, we are forced to select a specific implementation strategy. After detailing the specific requirements for each design pattern, we include an additional section that includes general requirements that force us to update AWSOME functionality.

In the following situation we outline essentially two sets of requirements. The first set describes the state we expect the model to be in before we can apply the transform.

Formally, these requirements may be thought of as preconditions necessary to apply the transform and are candidates for checking in a transform's *applicable* method. The second set of requirements detail what must be true about a model that has had the transform applied to it. Formally, these requirements may be thought of as post-conditions for each transform and are candidates for what a transform's *execute* method should do.

3.2.1 Singleton

The use of the singleton pattern will be deemed appropriate in the following situations:

- Only one instance of a class should ever exist..
- A point of global access must be provided for a particular object.

Before applying the singleton pattern to a model, we expect the following requirements to be satisfied:

- At least one class exists in the model to be used as a target for the transform, call it the *singleton* class.
- The *singleton* class must contain a default constructor.
- The *singleton* class does not contain a method named *getInstance*.
- The *singleton* class does not contain an attribute named *singleton*.
- In any place in which the model instantiates the *singleton* class, it must do so by using the *new* operator to allocate a new instance of the *singleton* class. This requirement is related to updating other classes to use the newly created *singleton* class.

The desired layout of a system implementing the singleton pattern is displayed in

Figure 3.4.

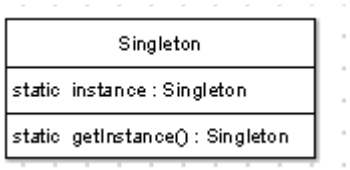

Figure 3.4 Singleton class diagram.

For this reason, we impose the following requirements as logical post-conditions to the transform:

- The *singleton* class must contain a private, static attribute that is the same type as the class itself. This attribute will store the only instance of the class.
- The *singleton* class must have its constructors changed to private, to prevent outside instantiation.
- The *singleton* class must also have a static method, call it *getInstance*, that has the logic to instantiate the private attribute, if necessary, and return the instance to the caller of the function.
- All places within the model that instantiated the *singleton* class via the statement "new singleton" have been replaced "singleton.getInstance()."

3.2.2 Abstract Factory

The use of the abstract factory pattern will be deemed appropriate if any of the following requirements are imposed on the system:

- Families of objects exist and the system must be constrained to use only objects belonging to the same family together.
- A collection of objects exists that should provide a common interface for interaction rather than expose concrete implementations.

We impose the following list of requirements as logical preconditions the model must support before the transform can be applied:

- Any concrete objects used in the model to be transformed must extend an existing abstract super class that defines a common interface that the subclasses implement.
- For each of the previously specified abstract super classes, they must contain an equal number of subclasses (the "themes" of that class). This requirement is to force symmetry and simplify the design.
- At any place in which the model makes use of a concrete subclass of an abstract class, it must constrain itself to only calling methods present in the abstract class. This requirement is to allow the transform to update usage in the model without requiring type casting any objects.
- At any place in which the model instantiates an instance of a concrete subclass, it does so via the *new* operator.
- Any method in a class in the model will make use of at most one "theme" of objects at a time. This requirement need not be explicitly checked, as it reasonable to assume that a designer would not, for example, mix a GUI related object with an object that makes use of the console.

The desired layout of a generic system supporting the abstract factory pattern is displayed in Figure 3.5.

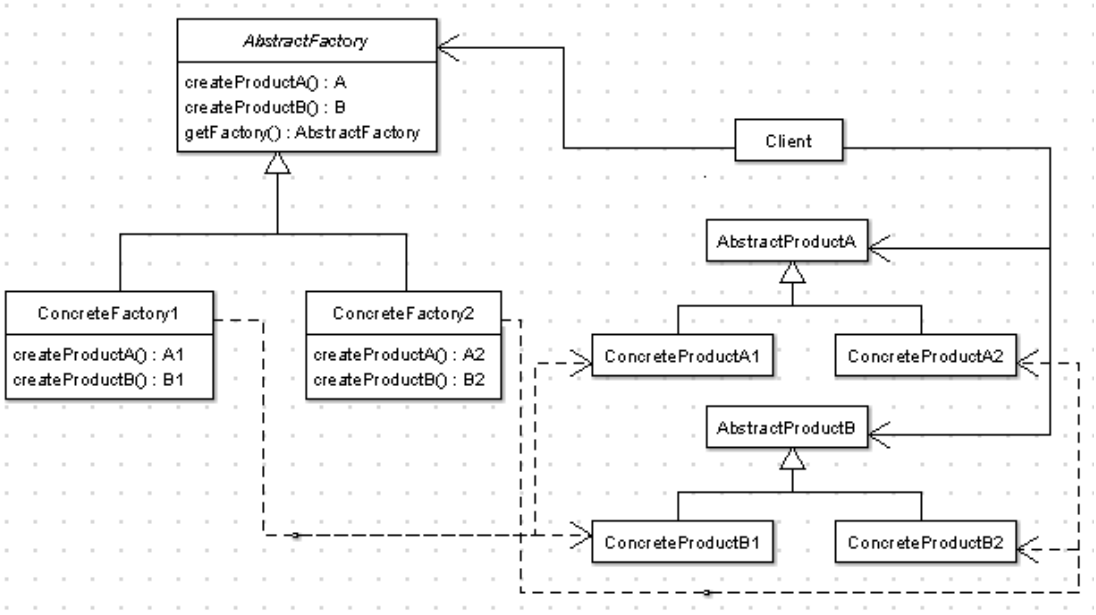

Figure 3.5: Abstract Factory class diagram.

In order for a system to successfully implement the shown abstract factory functionality, the following requirements are imposed logical post-conditions of the *execute* method of the transform:

- Each reference to a concrete *product* class in the existing model must be replaced by a reference to that *product* class's respective abstract super class. Method invocations regarding the references may remain unchanged as the super class specifies all abstract methods that encompass the required functionality implemented by the concrete subclasses.
- An abstract class must exist, call it *abstract factory*, that includes abstract methods for creating any type of concrete *product* that the client could ask the *abstract factory* to produce. In our abstract factory implementation, we opt to require that this class also contain necessary logic to provide access to concrete subclasses of itself to the user.
- Concrete subclasses of the *abstract factory* must exist that create concrete

products of the same "theme." These classes provide access to the created objects via the interface provided in the *abstract factory* class (typically a *createProduct* method).

- Any statements in the model in which a concrete *product* was being instantiated via the *new* operator must be updated as follows. The client must first acquire a *factory* that provides creation of the appropriate "theme" of objects. Next, the statement "new product" must be replaced with a call to the *factory's* associated *create* method to receive an instance of the product.

3.2.3 Factory Method

The use of the factory method will be deemed appropriate in the following situations:

- A class does not a priori know the class of objects it will need to create.
- A class would prefer that a subclass decides what type of object should be created.

We note that a model must satisfy the following requirements to be a candidate for the execution of the transform:

- For each concrete object to be transformed, an existing abstract super class must specify an interface implemented by the concrete object. We note that some objects may share the same super class. In addition, each concrete subclass must contain a default constructor.
- At any place in which the model makes use of a concrete subclass of an abstract class, it must constrain itself to only calling methods present in the abstract class. This requirement is to allow the transform to update usage in the model without requiring type casting any objects.

- At any place in which the model instantiates a concrete subclass, it does so via the *new* operator. This requirement is utilized in updating the model to use the pattern after it is applied.
- For each abstract *product* to be transformed, the model must not already contain a class named *productFactory* since the transform will create one.

A class diagram of a software system supporting the factory method is displayed in Figure 3.6.

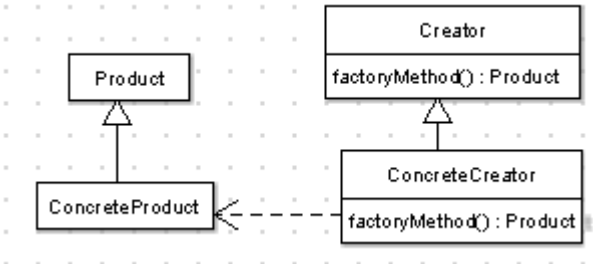

Figure 3.6: Factory Method class diagram.

In order for a system to fully support the factory method, the following requirements must be satisfied after executing the transform:

- A concrete creator class must exist with the naming convention of "productFactory" where "product" is the name of the abstract super class that the transform was applied to.
- *ProductFactory* must contain a method, call it *factoryMethod,* containing appropriate logic for providing access to concrete *products.*
- Any declaration in the model that is of a concrete *product* type must be changed into its abstract super class type.
- Any statement in the model in which a *product* is instantiated using the *new* operator must be replaced with a call to the *factoryMethod*.

3.2.4 Memento

The use of the memento pattern will deemed appropriate in the following situation:

- A class would like to store an image or state that could be restored at a later time without directly exposing the internal representation of the class.

The following requirements are posed as logical preconditions to apply the memento transform:

- A class that contains some attributes must exist in the model to be a target of the transform. We note that having no attributes would deem a class ineligible for this transform.
- The target class must contain methods for copying non-primitive attributes. In the case of collection types, we require that the class itself contain a method named "copyName" where "Name" is the name of the variable, designed to return a copy of the attribute. In the case of class types, we require that the class that matches the type exist in the model and have a method named *copy*, designed to return a copy of an instance of the class. This requirement prevents an issue of multiple classes containing references to the same object instead of separate copies of it.
- The target class must not already contain methods named *setMemento* or *getMemento* because the transform will create them.
- The model must not contain an existing class named "ClassMemento" where "Class" is the name of the target to apply the transform to.

The general layout of a system supporting the memento pattern can be seen in Figure 3.7.

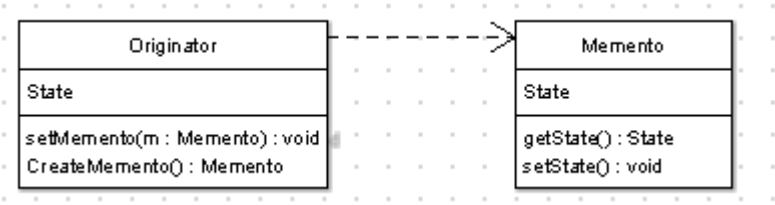

Figure 3.7: Memento class diagram.

In order to fully implement the memento pattern, a system must satisfy the following requirements after executing the transform:

- A separate class, call it *memento*, must exist that contains attributes necessary to store all data that represents the state of the original class, call it *originator*. In this case, we use the naming convention of "ClassMemento" where "Class" is the target class the transform was applied to. We note that we are forced to create a separate class because AWSOME does not allow the creation of private inner classes.
- *Memento* must have the capability to provide the *originator* with a complete view of its internal attributes so that it can access them to restore its original state. This access must be provided via *get* methods that return the private state of *memento*. Similarly, *set* functions must be provided to alter the existing state stored in *memento*.
- *Originator* must provide a *createMemento* function that enables a snapshot of the current state of the *originator* to be created as a *memento*. Furthermore, the originator must provide a *setMemento* method that uses a *memento* object to restore the *originator* to a previous state.

3.2.5 Observer

The use of the observer pattern will be deemed appropriate in the following situations:

- One or more objects may have a need or want to perform operations based on the current state of another object, and such objects should be notified when the state of that object changes.
- Changing an object may result in other objects being changed, but it is not a priori known which objects should be updated.

We require the following conditions be true of the model before the transform can be applied:

- In a system to be updated to support the observer pattern, classes named *Observer* or *Observable* must not already exist as the transform will create them.
- In the case in which we would transform an existing class to become an *Observer*, we require that the class not already inherit from a class, since multiple inheritance is not supported in AWSOME. In addition, we require that the class not already contain a method named *update*.

The general layout of a system supporting the observer pattern can be seen in Figure 3.8.

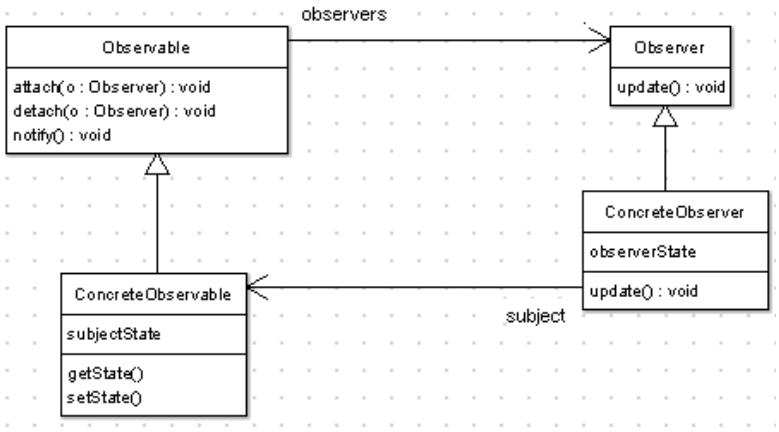

Figure 3.8: Observer class diagram.

We note that a typical implementation of an observer pattern would require specific action to be taken in the *update* method. However, we cannot automate the creation of the body of this method without the user directly specifying it. For this reason, we develop transforms that create an observer "shell" and allow new *observers* to be added. In order to fully implement our observer pattern, a system transformed must satisfy the following requirements:

- An abstract class that *observers* must conform to, call it *Observer*, must exist. This class must specify an *update* method of sorts that all subclasses must implement. The intent of the *update* method is to allow concrete *Observer* classes to take action when the object being observed changes.
- A class must exist for objects that are to be observed, call it *Observable*, which provides an interface for *attaching/detaching Observers* to the *Observable* class and for *notifying* them of changes.
- Any class that is to become an *Observer* must subclass the *Observer* class and declare the required *update* method.

3.2.6 Visitor

The use of the visitor pattern will be deemed appropriate in the following situations:

- An existing object structure must provide support for new operations without requiring that the existing classes be changed to enable the new operation.
- Many operations need to be performed making use of the existing class structure, but the operations do not belong in the existing classes themselves.

We impose the following requirements on a system as logical preconditions to

applying the transform:

- For a given class hierarchy that is to have the visitor pattern applied to it, it must be true that an existing abstract super class is present. This abstract class must have at least one concrete subclass present in the model to apply the transform to. The only requirement of these classes is that they do not already have a method named *Accept* as the transform will add it.
- The model must not already contain a class named "ClassVisitor" where Class is the name of the previously specified abstract class. We impose this requirement because the transform will add such a class.

The general layout of a system supporting the visitor pattern can be seen in Figure 3.9.

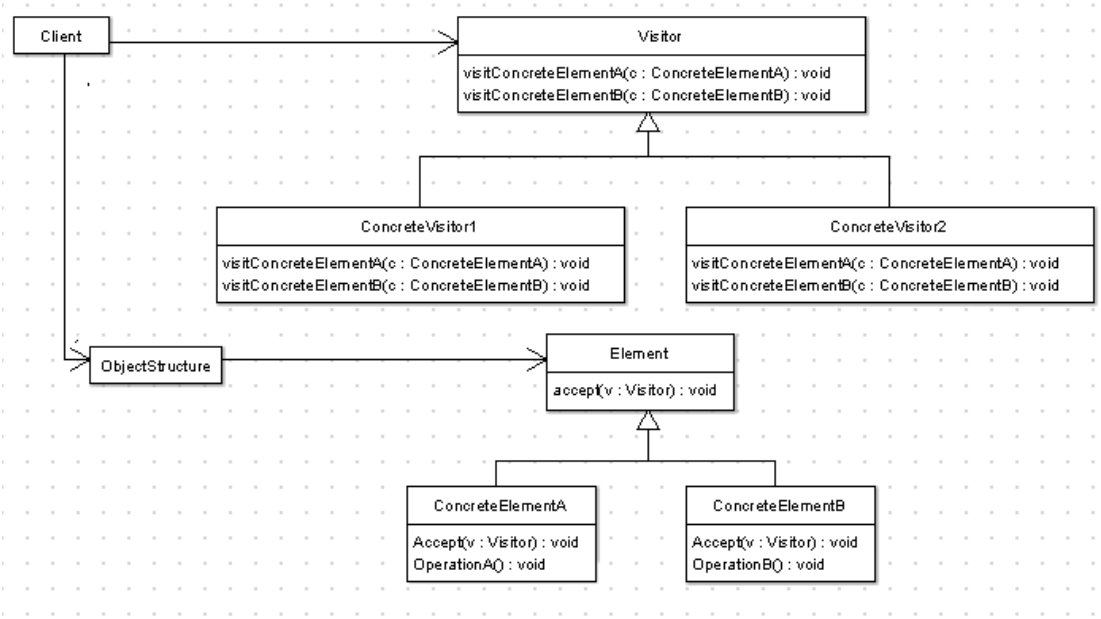

Figure 3.9: Visitor class diagram.

In automating the visitor design pattern application, we note that in a typical application of the pattern, concrete *visitor* subclasses would be created that perform some specific action in their *accept* methods. Since creating these classes would require

specific input from the user, we opt not to create any concrete *visitor* subclasses. With this decision in mind, we select the following requirements that must be imposed on a system that has been transformed:

- An abstract class, call it *visitor*, with the naming convention "ClassVisitor" where "Class" is the name of the abstract class specified as a target for the transform must exist.
- The abstract class representing the top level of a previously existing class used as the transform target, call it *Element*, must be still be present to define an interface all concrete classes must implement.
- *Visitor* must contain an abstract method, call it *visit,* that all subclasses must implement to specify new operations that can be applied to *Element* and its concrete subclasses.
- *Element* and all of its concrete subclasses must be updated with *accept* methods that contain logic to receive a request from a concrete *visitor* to supply the *visitor* with an instance of the *Element.*

3.2.7 Additional Requirements

While all of our previous requirements have specified preconditions and post-conditions for our transforms, we also have specific needs during the design phase that AWSOME will need to be updated to fulfill. While we previously noted many operations AWSOME must support to implement our transforms, such as allowing the creation of abstract classes, the requirements noted here were not supported by AWSOME before this thesis.

- Support must be added to represent the notion of static attributes in classes.

At bare minimum, this means that the classes involving the AST must allow an attribute to be represented as static. However, it would also be of great utility if the Java code generator and AWL generator also recognized the new static attribute functionality.

- The Java code generator must be updated to correctly translate into code the instantiation of classes via the *new* operator.
- The AWSOME libraries must provide support to find all the children of a given super class.
- The AWSOME libraries must provide support to generate a unique class name, albeit through the use of appending random characters to a name.

IV. DESIGN

Having laid a solid groundwork of the requirements expected to be fulfilled by applying each design pattern, the design and implementation of each can be worked in to the existing AWSOME.

4.1 Transform Class

At the highest level, a transformation in AWSOME is actually a first-class object. In that regard, an abstract super class defines precisely what methods a concrete transformation class must implement. The main functions that must be implemented to do the transformation work include *applicable* which returns a *boolean* indication whether or not the transformation can be applied to the model, and *execute* which actually modifies the model to apply the transform and returns a *boolean* indicating success. It should be noted that the *applicable* method must be overridden in subclasses because it is declared static in the super class to allow querying the transform without having to create an instance until the pattern is to actually be applied. Furthermore, every *execute* method will call *applicable* to confirm that the transform can be applied before making any other changes. If the transform cannot be applied, the *execute* method will immediately return *false*. Other functions included in the transform class such as *replay* and *undo* are geared toward reloading existing models and reversing the application of a transformation. Future work should focus on implementing these functions for our transformations, but will be considered beyond the scope of this thesis.

Once a complete transform class has been implemented, it must be anchored to the existing AWSOME tool. The approach of this thesis is to introduce a panel that is

responsible for housing buttons to apply each implemented design pattern transformation as well as provide an area to show or query any relevant information for a given transformation. The detailed discussion of this panel is delayed until Chapter 5.

Although AWSOME contains many existing transforms and library methods, the introduction of design patterns to the system necessitates additional functionality. Throughout the design chapter, specific problems that require the introduction of new library functions will identify the new functions by name, but will postpone their definition until the end of this chapter. In addition, any deficiencies in the AWSOME framework that require modification to support our patterns will be noted, though solutions will be postponed until 4.14.

4.2 Singleton

4.2.1 Introduction

The singleton pattern exists to ensure that only one instance of a particular class exists at a time and to control access to the instance. Given this fact, it is possible for any class to be transformed into a singleton.

4.2.2 Transform Inputs

In order to apply the singleton pattern, we require only a specific class. For this reason, we need only present a list of classes in the existing AST and allow the user to select one to apply the singleton pattern to.

4.2.3 Applicable

The *applicable* method begins by first confirming that the argument passed is actually a class, as the transform can only be applied to an entire class. After confirming this, *applicable* moves on to check for a default constructor as it is necessary to instantiate the

singleton object that will be added to the class. Next *applicable* confirms that no attribute named *singleton* already exists as one will be created in the *execute* method. Finally, the class's methods are searched to confirm that the class does not contain a method named *getInstance* that has an empty argument list. If any of these previous requirements are found to be violated, the function will return *false* to indicate the transform cannot be applied.

4.2.4 Execute

Execute begins by creating a private, static attribute in the class of the same type as the specified class with the name *singleton*. Once the attribute has been created, we continue the *execute* method by creating a new public method named *getInstance* with a return type set to the same as the specified class. Inside of this method, we create an *if* expression that tests if the *singleton* variable currently has the value of *null*. If it does, a new instance of the class specified is created using the default constructor and assigned to the variable *singleton*. The *if* expression contains an empty *else* clause as there is nothing to do if the object already exists. After this expression has been created, we next add a return statement that returns the *singleton* variable. Finally, we convert the class's default constructor to private, to prevent outside instantiation.

4.2.5 Existing Model Updates

Once we have successfully updated a class to implement the singleton pattern, we must modify the entire AST to make use of the changes. Rather than describe this process here, it is outlined in the next section as a separate transform. One noteworthy limitation posed by the language of AWL that is relevant to this transform is the fact that AWL does not contain syntax to indicate an attribute is static. In other words, while the

transformed AST can be written back to an AWL file, the singleton attribute will not be marked as static and will have to be turned back into static if the AWL file is loaded back in. This issue is further addressed in 4.14.

4.3 Singleton Usage Transform

4.3.1 Introduction

While there is clearly value in adding support for the singleton pattern to a class, this causes problems if it is the only change made in the model. The primary problem hinges on the fact that since the constructor is now made private, any previous place where the class was instantiated will no longer function. In order to remedy this, we create a transform that will update the other parts of the model to use the singleton class.

4.3.2 Transform Inputs

In order to properly execute this transform, we require the entire AST that has possible usage of the singleton class. Furthermore, we also need an input in the form of the class that has become a singleton. With both of these inputs, we can search the AST and find the places where it uses the singleton class.

4.3.3 Applicable

At its heart, the *applicable* function is essentially a check that the singleton pattern did precisely what its *execute* was designed to do. *Applicable* begins by checking the specified class for a method named *getInstance* as well as a default constructor. If neither function is found it returns *false* to indicate the transform cannot be applied. Finally, the class is checked for an attribute that is static, has the same type as the class, and has the name *singleton*. If not found, the function again returns *false*, otherwise the transform is applicable.

4.3.4 Execute

While the main idea of the *execute* method is to simply replace each call to the default constructor using the *new* operator with a call instead to the singleton class's *getInstance* method, it requires significant work to find these places. To begin, we must search every method in each class of the passed AST. Within each class, we must search every method for the usage of the *new* keyword. However, it does not end here due to the fact that AWSOME treats conditionals (*loops*/*if-else* statements) as first class objects. Since conditionals are first class objects, we recursively search each one for nested *new* and conditional statements. However, once we have found all these instances, we update them with the call "singletonClassName.getInstance" where "singletonClassName" is the name of the singleton class.

4.3.5 Existing Model Updates

Once we have replaced all statements in the specified AST that used the *new* operator to allocate an instance of the singleton class with the call to the singleton's method, the model will fully support the design pattern. While it is possible that an engineer could add to the model and add new places where an instance of the singleton class is required, there are two solutions to said problem. The first involves the engineer simply using the *getInstance* function instead of the *new* operator while the second requires only that the transform be run on the AST again (thus allowing the designer to use the *new* operator instead of *getInstance*).

4.4 Abstract Factory

4.4.1 Introduction

The abstract factory pattern relies on the notion that certain concrete objects used by a

system may fit into various groups based on the "theme" of the objects. The intent of the pattern is to ensure that the same "themes" of objects are instantiated together, rather than mix the types.

4.4.2 Input

In order for this transform to be executed, we require both the entire AST as well as a specification of the classes and their themes. However, automating the process of detecting into what groups an existing set of objects should be partitioned proves to be quite a challenge. While it may be possible to separate concrete objects into groups based on underlying implementation details such as references to a graphical user environment (GUI) versus console input/output and abstract super classes, this thesis takes a different route. Clearly the designer knows best which objects fall under a similar theme, so it will be the role of the transform user to partition the classes into groups before calling the transform. In order to simply the task of querying the user, a library function named *returnChildrenNames* has been created and is detailed in 4.15. The transform expects a list of lists, where the first element of each list is the super class of a particular group and each subsequent element in the list is the 1st theme, the 2nd theme, and so on as concrete subclasses. For example, we may have an abstract tic tac toe class, call it *TOE*, that contains all abstract methods that must be implemented by a subclass in order to have a complete game. In addition, we have two concrete subclasses of the tic tac toe class, *GUITOE* and *ConsoleTOE*, with the first implementing a graphical version of the game and the second a console based version utilizing ASCII art. If we were to use these classes as input to our transform our list of lists would contain one list as the first element with the contents of *TOE*, *GUITOE*, and *ConsoleTOE* in that order.

4.4.3 Applicable

The *applicable* method begins by ensuring that the passed input is precisely what the transform requires. Namely, it makes sure that the input is a list of lists, with the first element of each list being a parent class and each subsequent element in the list inheriting from the first element. In doing so it makes sure that there are the same number of classes for each particular theme, as it simplifies creating the methods for getting each particular object. Finally the *applicable* function ensures that each of the specified classes has a default constructor as it will be necessary to instantiate them in the methods that create instances of the classes. If all these tests are passed, the function will return *true* indicating that the transform can be applied.

4.4.4 Execute

To begin, we create an abstract class whose name is randomly generated by calling the *makeUniqueClassName* function (detailed in 4.15) and passing it the base *String* of "AbstractFactory." Now, for each of the abstract product classes passed to the function, functions will be added to the class with the naming convention of the word "create" concatenated with the name of the abstract class. Each of such functions will have its return type set to abstract class being considered.

With these functions implemented, it is necessary to shift attention to creating the concrete factory subclasses of the *AbstractFactory* class. Each of these classes' parent class is the *AbstractFactory* class and each name is determined by passing the *makeUniqueClassName* function the base *String* "ConcreteFactory". Next, functions are added to the class to implement all functions specified in the abstract parent class. The body of these functions will be set up to create new instances of the associated product

type and theme and return a reference to it. This will be accomplished by using the list of classes passed to the function and the order of the list as previously described.

Once all the subclasses are created, they can be added into the existing AST and the abstract factory class can be completed. To complete the implementation, we will create a static function named *getFactory* that returns a concrete factory subclass. The function will take an integer argument corresponding to the theme of objects desired and use a series of if statements to check if the number matches an existing theme and return a reference to a factory that matches the type (if it exists). Since we do not know a priori the name of a theme, we will instead use the randomly generated concrete factory names in the logic to decide what factory to give the function caller. With this step completed, the class itself can be added into the existing AST.

4.4.5 Existing Model Updates

After these classes have been created, the only remaining action required by the transform is to update the AST to make use of the abstract factory pattern implemented. Rather than describe this action here, it is detailed in the next section which presents a transform to automatically process this update.

4.5 Abstract Factory Usage Transform

4.5.1 Introduction

Although adding support for the abstract factory pattern to an existing model is useful in the regard that future updates to the model can take advantage of it, there is still more that could be automated. In attempting to further automate the application of this pattern, we create a transform that explores the model and updates it to make use of the newly created abstract factory.

4.5.2 Transform Inputs

In order to apply the usage transform, we again require the AST itself as well as the list of lists from the previous transform. However, we also require that added to the very end of this list is another list that contains the newly created abstract factory class as the only element (since we have no way of automatically detecting the name of the factory).

4.5.3 Applicable

The *applicable* method begins by confirming that the input is precisely what is expected in 4.5.2, and returns *false* if it finds it is not. Indeed, this check is nearly identical to the one from 4.4.3, and the only additional check we have is that the last element is the abstract factory class. In fact, we expect that this list be in the same exact order as the previously supplied one. The next job of the *applicable* function, arguably the most important one, is to search the entire AST (excluding classes passed in or classes related to the factory we are updating usage for). In this search, we look at all method calls of the form "object.MethodName()" where the "object" is one of the concrete products passed as input. In any situation where this occurs, we must check that the parent class contains a method named "MethodName," otherwise we are calling a method that only exists in the child class, which means we cannot change the "object" type to the super class (or we would have to type cast it). If the *applicable* does not encounter this situation in any class, it returns *true* to indicate the transform can be performed.

4.5.4 Execute

We begin *execute* by searching classes in the AST (minus the product and factory related classes) for instances in which the *new* operator is used to create a concrete product object. While seemingly a simple search, we must recursively search all

conditionals (loops and if/*else* statements) for nested statements since AWSOME treats conditionals as first class objects. For each such class that uses the *new* operator to instantiate a concrete product, a private attribute with the type set as the abstract factory and whose name is the same as the abstract factory but in all lowercase, to prevent potential name clashes, is created. Now for each place the new operator is used to create a concrete instance of an object class, we will first check which "theme" the object fits under. This is trivially done by looking through the input list of classes and locating said class. Once this is known, we will assign the abstract factory attribute a new concrete factory by calling the static *getFactory* method provided in the abstract super class. Deciding which integer to pass it is done by noting the index of which class in the input list matched. It should be noted that this statement to update the factory to the correct type is inserted at the beginning of the function we are modifying as it is expected that at a function would only use one type of "theme."

Once this task has been accomplished, we will update the actual line in which the *new* operator was being called. In this case, we will check the super class of the concrete product previously being instantiated and replace the side of the assignment containing the *new* operator with a call to the abstract factory attribute's "createClass" method, where Class is replaced with name of the product super class. With this step completed, all we finally change the type of the variable on the left hand side of the assignment to the product super class type.

4.5.5 Existing Model Updates

While the usage transform takes an existing model and updates it to make use of the abstract factory, it is expected that this transform would be called right after applying the

initial pattern transformation. As such, it does not lend itself well to being called again later, after an engineer has made some other changes, to again update usage in the model (since it creates the attribute). While we could add an additional transform that updates the usage again in the event that the engineer made changes, this thesis does not opt to make such a transform. The primary reason this is not done is that once the initial usage transform has been completed, any new changes to the model should adhere to using the abstract factory that was created via the initial pattern transform. In other words, it shouldn't be the case that a statement that could use the factory, but doesn't, is added to the model.

4.6 Factory Method

4.6.1 Introduction

The factory method exists to specify an interface that can be used to decide which type of class to instantiate. The factory method allows logic to be used to pick which class type should be instantiated and allows for subclasses to specify different implementations.

4.6.2 Transform Inputs

In order to transform the model to support the factory method, a list of class names must be specified that are potential candidates to be returned from the factory method itself. The common super class of each of these classes is also needed, but this information can be collected by checking the super class of the passed classes.

4.6.3 Applicable

The *applicable* method begins by verifying that the input is in fact a list of classes that exist in the current model. Next, the method checks that all the specified classes have a

common super class that exists in the model. In addition, the classes are checked for default, no-argument constructors, as they are required to properly instantiate the classes in the factory method. Finally, the model is checked to ensure that it does not already have a class with the naming convention of the common super class name concatenated with the word "Factory," as the transform will create such a class.

4.6.4 Execute

The *execute* method begins by extracting the super class name from the supplied class arguments and creates a new class with the super class name concatenated with the word "Factory." Next, a *boolean* attribute is added to the new class for each passed class, which will be used to indicate which type of object the factory should instantiate. To make use of these flags, a function is added to the new class entitled *getInstance* with the return type of the common super class. The body of this method is filled out with a series of *if* statements that essentially check which of the flags is *true* and returns the corresponding object type that matches the flag. Finally, in order to set up the type of object the *getInstance* method should return, a series of set methods are created for each of the passed class names with the naming convention of the word "use" concatenated with one of the passed class names. When this function is called, it sets the corresponding class flag to *true* and all others are set to *false*.

4.6.5 Existing Model Update

While adding support for the factory method has the benefit that future updates made to the model by an engineer can utilize the pattern, we can also update the existing model directly after the transform is applied. In this regard, we may find places in the model that are candidates for updating and change them to actually use the pattern via the

transform detailed in the next section.

4.7 Factory Method Usage Transform

4.7.1 Introduction

Adding the notion of a factory method to the existing model certainly has merit, but we note that we can also force the model to actually use the factory method. In doing so, we find that we can automate the process of finding statements in the model that are candidates for updating to use the previous transform. Once this transform is complete, the engineer will not need to make any changes to the existing model, but rather will only need to use the factory method in future updates to the model.

4.7.2 Transform Inputs

The input required to this transform is precisely what is expected in 4.6.2. That is to say, we again require a list of classes that inherit from a common super class. While one may intuitively wonder why the factory class is not also required, the transform itself is capable of finding the factory via the name of the super class of the passed classes.

4.7.3 Applicable

Like any other *applicable* method, this method begins by ensuring that the input is of the format required. Next, we essentially check that the factory method transform has been applied by searching for a class with the naming convention “nameFactory,” where name is the name of the super class common to the list of input classes. We then move on to the important task of searching all classes in the input AST (minus the input classes and the factory class) for places in which we have statements of the forum ”object.functionCall().” For each such instance, we check if the object is one of our input classes, and if it is, that the “functionCall” is a function present in the parent class. This

check is required since we will change the type of the object to the super class and would have to type cast the function call if it was not a method in the parent class. Next we ensure that the classes do not have an attribute with the naming convention "theSuperClassNameFactory" where "SuperClassName" is the name of the input classes' super class. Finally, we ensure that each class has a default constructor since we will initialize the "theSuperClassNameFactory" attribute in the constructor.

4.7.4 Execute

In order to update the model to utilize the newly added factory method pattern, we must again recursively search all classes in the input AST (minus the classes in the input list and the factory class) to find all statements in which we use the *new* operator to instantiate one of the input classes. For each such class, we will add a new attribute named "theSuperClassNameFactory," where "SuperClassName" is the name of the input class's parent class. The type of the attribute is the type of the newly created factory. With this complete, we then go to each function that includes a statement to instantiate one of the input classes and replace the part of the statement with the *new* operator with a call to the factory attribute's *getInstance* method. For each such method, we then insert a statement at the beginning of the method that has the effect of setting up the factory to return the right kind of object. This is done simply by looking at the type of the attribute on the left of the assignment statement and calling the factory attributes "useType" method where "Type" is the object's class type. We do this only once assuming that at most a function would use only one "type" of a particular object.

With the previous steps completed, we then change the type of the value on the left side of the allocation statement to match the super class type. Finally, once all such

statements in the class have been transformed, we initialize the factory attribute in the default constructor of the class using the factory's default constructor.

4.7.5 Existing Model Updates

With the successful execution of this transform, no further updates are required to the model in order for it to fully utilize the factory method pattern. However, in the future, an engineer may add additional functionality to the model that may make use of the factory method. While another transform can be provided that takes a modified model and again updates it to utilize the pattern, this thesis does not choose that route. Instead, it is assumed that once the pattern has been applied, any changes made by an engineer will take into account the pattern and utilize it, as there would have been no point in including the pattern if it is never actually used.

4.8 Memento

4.8.1 Introduction

The memento pattern is used to create a class capable of encapsulating data members of an existing class that can be used to restore said class to a previous state. For this reason, any class with internal data members is a plausible candidate for the memento pattern to be applied.

4.8.2 Transform Inputs

In the case of the memento transform, we require as input a single class that requires a memento class to be created for it. For this reason, we require the user to simply specify a single class from the existing AST and then pass the chosen class to the actual transform.

4.8.3 Applicable

The *applicable* method begins by ensuring that the input passed in is actually a class as the transform can only be applied when an existing class is specified. The next check performed is that the model does not already contain a class with the naming convention of "ClassMemento" where "Class" is the name of the class the user selected. While typically a memento would be implemented as a private inner class in the chosen class, AWSOME does not support this feature and as such we must make sure that a memento has not already been created for the class. In addition, the *applicable* method will check that methods named *createMemento* and *setMemento* do not already exist in the selected class as they will be added by the transform.

After ensuring that the target to apply the transform to is a class, the *applicable* method then ensures that there are actually attributes in the class as a class with no data members has no state to store. Once we have ensured the existence of attributes, we then evaluate the type of each attribute and for each that is not of the basic type integer, *boolean*, or *double*, that is to say it is an instance of some class or collection type, we ensure that some sort of method to copy the data exists. In the case of collection type such as an array, the class provided as input will be checked for a method named "copyNameOfAttribute" where the "NameOfAttribute" is the name of the array attribute in the class. In the case of an attribute that is an instance of another class, that attribute's class will be checked for a method named *copy* whose intent is to produce a copy of the object. This requirement is necessary to make sure that the memento can receive actual copies of objects and not just references to them as otherwise the memento and class will have the same object.

4.8.4 Execute

The *execute* method begins by creating a new class in the AST with the name "ClassNameMemento" where "ClassName" is the name of the class the user supplied to create the memento. Next, the method continues by making a copy of each attribute in the specified class in the new memento class, but changes each to private visibility. Once all the attributes are created, the *execute* method creates a constructor in the memento class that takes all attributes of the class as parameters and sets the private class data members to the values passed to the constructor. This is achieved by making use of a function created based on the XformArjun3 transform with some small modifications [14].

With the constructor created, the *execute* method then moves on to create *get* methods for all the attributes. While typically the memento is created as a private inner class inside of the class the memento stores data for, limitations in AWSOME prevent this approach from being used. As such, we must provide ways to access the data in the memento through *get* methods. Furthermore, although a client should create a new memento through the constructor and should not be able to change parts of an existing memento, we also create *set* methods to update the state of the memento. We are forced to make this concession as at the time of this thesis's writing, a call to a multiple argument constructor will not be turned into syntactically correct Java code by the code generator. The functionality of creating the *get* functions is accomplished by first creating new methods with the naming convention of "getAttributeName" where "AttributeName" is the name of one of the attributes. Next a body for each such *get* is created out of an assignment where the left hand is the name of the function and the right

hand side is the name of the class attribute (this is the equivalent of a return statement in the AST/AWL). The *set* methods are created in a similar way by adding functions to the class with the naming convention of "setAttributeName." In addition, the bodies of the function consist of an assignment where the internal class variable is assigned the value passed to the method as an argument.

After the previous operations have been completed, the *execute* method moves on to modify the user specified class to provide support for the new memento. To begin, *execute* creates a new method named *setMemento* that takes as an argument the newly created memento class. In this method, a series of assignment statements are used that call the passed memento's *get* methods and assign the result to the class's internal attributes. No copy method is used in this case as a memento's internal attributes cannot be changed; a user would have to declare a new instance of a memento in order to change them, thus the reference problem does not arise. Next, we create a *getMemento* function that constructs a new memento object and returns it. This is achieved through making copies of each attribute (although we do not make copies of primitives such as integers) and storing each attribute in local function variables. We then create a local memento object and call the set method for each of its attributes, passing the local variables each time. Once this is completed, the function simply returns the newly created memento. While typically a memento class would return this type as some super type class (such as Java's "Object" class), a notion of a super class of all classes does not directly exist in AWSOME and as such we are forced to return the memento as a memento object type.

4.8.5 Existing Model Update

Once the memento design transform has updated the model to provide memento

support for a particular class, no further modifications are made to the AST. Typically a memento is used in situations where an undo operation is desired and as such the designer would have other classes making use of the memento class As such, only the designer would truly know when mementos would need to be created to save the state and when they would be used to restore a state. For this reason, we opt to allow the designer to make use of the newly created memento support, but make no a priori decisions of where this use would appear in the existing model.

4.9 Observer

4.9.1 Introduction

The observer pattern is used to allow an object to update other objects on changes of some object's state automatically. For this reason, this pattern is appropriate when existing classes should be informed when some other class changes its state. Like previous transforms, this transform is split into two independent transforms, the first of which simply adds observer support as a shell and the other which allows the creation of new observers.

4.9.2 Transform Inputs

Unlike previous transforms, the input to this observer transform is minimal. In fact, we require only the AST passed as input as the transform will create the required classes independent of any other factors.

4.9.3 Applicable

As expected, the *applicable* functional does not contain a large number of input guards. In this case, we require only that the model does not already have classes named *Observer* or *Observable*, since the transform will create these classes.

4.9.4 Execute

Execute begins by creating an abstract class named *Observer* with an abstract method named *update*. Next, an abstract class entitled *Observable* is constructed with three abstract methods: the *attach* method which takes an *Observer* as input, the *detach* method which takes an *Observer* as input, and the *notify* method which takes no arguments.

4.9.5 Existing Model Update

As the observer pattern is intended to allow the programmer to specify operations a class should take when the state of the object it is watching changes, it is difficult to make the subclasses of the *Observer* class perform a specific action in their *update* method. Furthermore, in order to complete the *Observable* class, the class must receive a storage container of some type to hold all of the *Observer* objects as well as have its *attach* and *detach* methods updated to access the storage and modify it. Hard coding a particular storage type in the model is not done to allow flexibility in the choice of which type the developer wants to use. In addition, the *Observable* class's *notify* method must be updated to go through the implemented storage type and call each individual element's *update* method. Again this is left out as varying storage types would require different techniques of negotiating the list.

4.10 Observer Add Class Transform

4.10.1 Introduction

While the previous transform sets up a shell that can be expanded upon to define behavior for the *Observers*, it may be the case that some classes should be set up to become *Observers*. In the case of this transform, we take a specified class and set it up to become an *Observer*.

4.10.2 Transform Inputs

As is common with all transforms, our first input required is the entire AST that is to be modified by the transform. The second piece of input that we require is a list of class names that we would like to turn into *Observers*.

4.10.3 Applicable

While the previous transform had relatively simple requirements to be applied, changing a new class into an *Observer* requires more condition checking in *applicable*. To begin, existing *Observer* and *Observable* classes must exist in the model. In the case of the *Observer* class, it must have a method entitled *update* whereas the *Observable* class must contain methods named *update*, *attach*, and *detach*. Essentially, we do a formal check that the previous transform to create the shell was actually applied. The *applicable* method then checks the list of classes provided as input to ensure that there are classes to apply the transform to. Finally, *applicable* ensures that the classes to transform actually exist in the model, have no super class, as multiple inheritance is not supported, and that the class does not already contain a method named *update*, as this method will be added.

4.10.4 Execute

Execute begins by setting each of the input classes' super class to the *Observer* class previously created in the model. Next, the transform adds a public method named *update* to each of the input classes since the parent class declared it as abstract. With these two steps completed, the list of input classes have now been transformed into *Observers*.

4.10.5 Existing Model Update

While the execute method does the work of turning each of the input classes into

Observers in the sense that they subclass the *Observer* class and include the *update* method, there is more that could be done. Namely, the engineer must now alter the model to actually do some particular action in the newly created method. Since we do not know a priori what action an *Observer* should take based on the change of some *Observable's* state, we leave this decision up to a human designer.

4.11 Visitor

4.11.1 Introduction

The visitor pattern is used to introduce new operations to be performed in an existing class hierarchy. As such, any set of a concrete classes inheriting from a common super class is a feasible target for the visitor transform. In designing the transform to apply the visitor pattern, three separate transforms are created. The main transform is detailed here, while the other two minor transforms are detailed in the following two sections.

4.11.2 Transform Inputs

In the case of the visitor pattern, we require the AST to be modified as well as an existing set of classes that inherit from a common super class. Since AWSOME does not support the idea of multiple inheritance, the transform requires only that the user specifies the classes that inherit from a common super class. The transform can deduce the common super class by checking what class they inherit from. In addition, the transform will require the user to specify a name for the newly created abstract visitor class to be created which will be expected to be the last element in the list of classes passed as input.

4.11.3 Applicable

The *applicable* method begins by ensuring that specified arguments are what the transform expects. In this case, it is expected that a list is specified whose first elements

are classes and whose last element is a String. In addition, the method ensures that the list has at least two elements as it would be useless to apply the transform without at least one class and a name for the visitor class. Once these checks are completed, the *applicable* method ensures that all specified classes inherit from the same common super class. Next, the *applicable* method examines the existing AST to discover if a class name that is formed by concatenating the user specified name with the word "Visitor" already exists. If it does, the *applicable* returns *false*. Finally, the method examines all methods defined in the specified classes to ensure that a method entitled *Accept* that takes as an argument the specified class name concatenated with the word "Visitor" does not exist. In addition, the super class of the subclasses is also examined for this constraint. If it is not found to exist in any of the classes, the *applicable* method returns *true*.

4.11.4 Execute

Execute begins by creating a new abstract class with its name derived from the user specified name concatenated with the word "Visitor." Several abstract methods to be implemented by a concrete visitor are then created for each user specified class with the naming convention of word "Visit" being concatenated with the name of the class. Each of these methods take as a single argument a variable of the type of the corresponding user specified class. Upon completion, each of these methods is added to the newly created v*isitor* class.

With the *visitor* class created, the attention is then turned to the user supplied classes. For each class, a new method is created named *Accept* that takes as a single argument a variable with the type of the newly created *visitor* class. A body for this method is then created that simply makes a callback to the v*isitor* class, passing itself as an argument.

After all of these methods have been added to the classes, the super class that each class inherits from is also given the same method, with the exception that is an abstract method with no body.

4.11.5 Existing Model Update

In the case of the visitor pattern, once it has been successfully implemented into the existing model, no further changes are made to support the addition. The reasoning behind this is that the visitor pattern exists to facilitate new operations that will be created on the existing class inheritance hierarchy. For this reason, it is left up to an engineer to create concrete subclasses of the v*isitor* class that accomplish new functionality.

4.12 Visitor Attribute Addition Transform

4.12.1 Introduction

While the initial visitor transform sets up the system to support the pattern, in some cases it is desirable for the *accept* and *visit* methods to take additional arguments. The goal of this transform is take a new argument type and name that can be passed along to these methods and update the visitor pattern in the model to utilize the change.

4.12.2 Transform Inputs

As with all transforms, the first input required by this transform is the AST that is to be modified. The second argument is a list of four *Strings*: the name of the v*isitor* class (created by previous transform), the name of the super class in the inheritance hierarchy of the classes that get visited, the type of the new attribute to be added, and the name of the new attribute. The order of these arguments must be preserved as presented here.

4.12.3 Applicable

The *applicable* method begins by ensuring that the input list is comprised of four

Strings. Next, the model is checked to ensure that both of the classes specified exist and that the type specified for the new attribute is declared somewhere in the model. Once these checks are completed, the *applicable* method returns *true* to indicate the transform can be applied.

4.12.4 Execute

While the *applicable* method was rather simple, the *execute* method performs a significant amount of work to update the input AST. To begin, the transform goes through the *visitor* super class and all its subclasses searching for the methods of the naming convention "VisitClassName," where "ClassName" is one of the classes we could visit. For each such method, a new argument is added that is of the user specified type and name. With this complete, we move our focus on to the classes we actually visit and their abstract parent class. As before, we search for all methods that are named *Accept* and simply add the user specified attribute and type as an argument to them. However, we only update *Accept* methods that take our *visitor* super class as an argument, since other ones could belong to a different type of visitor.

Although that was enough to update the *visit* methods, we must also change the function call backs in the *accept* methods. In this case, we replace the line that had previously read "visitor.VisitClassName(this)" (where ClassName is one classes we could visit) and replace it with "visitor.VisitClassName(this, null)." Rather than attempt to instantiate an object that matches the second argument type, we simply pass no instance, that is, a *null*.

4.12.5 Existing Model Updates

While simply passing *null* as a second argument to the *visit* method call may appear as

an unnecessary simplification, it is not necessarily so. Typically a second argument is passed into *visit* methods to allow additional information to be carried over to the *visitor* object. Since we do not have any way to automatically deduce what type of information should be sent, we opt to insert a *null*, thus leaving it up to an engineer to decide how an object of the appropriate type should be instantiated and passed. This approach is reasonable as it is follows our technique in which we left it up to the user to create *visitor* subclasses and decide what behavior they should take given the information received from calling a class's *accept* method.

4.13 Visitor Super Class Transform

4.13.1 Introduction

In creating the *visitor* classes, a new transform that proved to be of use surfaced. This transform, while relatively simple, was used to create an abstract parent class and make a specified set of classes inherit from it. Rather than shove this task into the visitor transform itself, it was deemed that this could be a useful transform for future work and was instead turned into its own transform.

4.13.2 Transform Inputs

As expected, the transform takes as one input the AST to be modified by the transforms. The second argument is again a list, with the first elements being classes existing in the model and the last argument being a *String* name of a class. The last argument will be used to create an abstract super class whereas the first arguments will have their super class updated to the newly created class.

4.13.3 Applicable

To begin, *applicable* ensures that the input has the required format, and returns *false* if

it does not. Next, the specified AST is checked to ensure that the specified classes actually exist in the model. Next, we make sure that none of these classes already have a parent class, as multiple inheritance is not supported in AWSOME nor would we like to disrupt an existing hierarchy. Once that check is complete, we determine whether or not there is already a class in the model that shares a name with the last argument in the input list. Since we will be creating a class with this name, it cannot already exist in the model. If *applicable* finds that none of these constraints are violated, it returns *true* to indicate that we can apply the transform to the model.

4.13.4 Execute

Execute begins by creating a new abstract class that has the name as specified by the last argument in the input list. With the class created, each of the other input classes have their super class set to the newly created class.

4.13.5 Existing Model Updates

Given that the nature of this transform is used more as a utility to create a new inheritance hierarchy give a list of classes with no parent class, its effects are self contained. That is to say, the model does not need to make any changes to utilize this new update. While there are a multitude of directions we could take from here, such as allow new methods to be added to the super class and force all subclasses to specify them or change all objects defined in the model with a subclass type to the super class type, this transform was only created to set up classes for the visitor pattern. As such, future work may build on what this transform does, but its intent for this thesis is kept simple.

4.14 Modifications to Existing AWSOME Framework

As several of our design outlines mention, there are some limitations in AWSOME

that prevent our transforms from doing exactly what we desire. In this section, we outline the changes we have made in order to make our transforms work as specified. In some cases, due to time constraints, we were not able to update AWSOME to fully support a given requirement.

Our first change is in regard to a design decision made by the original architects of AWSOME. While the foresight to provide static functions was included initially in AWSOME, the notion of static attributes was not. The lack of static attributes poses a serious problem for several of our design pattern implementations. Fixing this problem requires modifying four parts of AWSOME: the AWL parser, the *WsAttribute* class itself, the *WsOutlineVisitor* class itself, and the Java code generator. We note that changing the AWL parser is beyond the scope of this thesis and are forced to make the concession that writing our transformed ASTs that create static attributes back to AWL will cause them to lose the notion of static. In the case of *WsAttribute*, we simply added support to type an attribute as static, thus making the AST itself recognize static attributes. In modifying the *WsOutlineVisitor*, we added support to force it to print whether or not each attribute was statically typed. In the case of the Java code generator, we modified it to recognize a static typed *WsAttribute* and write it to Java code using the keyword static. These collective changes make it possible to at least demo the fact that our transforms utilize statically typed attributes.

The next change is related again to the Java code generator. We note that initially AWL's *new* operator was turned into an array declaration when parsed into Java code, even if our intent was to instantiate an object. Given that AWL is an object-oriented language, it seems reasonable that object instantiation should correctly translate to Java

code. For this reason, we modified the Java code generator to recognize when the *new* operator is being used in conjunction with a *WsClass* and correctly turn it into a Java instantiation of the form "new ClassName()" where "ClassName" is the name of the class to instantiate. However, we did not update this to handle a multiple argument constructor and thus require classes to have a default constructor if they are to be instantiated. Fortunately, a transform exists in AWSOME that allows a default constructor to be added to a class.

4.15 Library Functions

In order to facilitate the design of some of the transforms created in this thesis, we introduce some library functions. Rather than hard code them into our transforms, we have added them to AWSOME libraries in hopes that they will be useful in future work.

1. *returnChildrenNames()*

   This method was added to *ToolUtils.java*. The function takes as input an entire AST and the *String* name of a class in the model and will return a *vector* of *Strings* representing all classes that inherit from the specified class.

2. *makeUniqueClassName()*

   This method was added to *XformUtils.java*. The function takes two arguments: a *String* to be used as a base for the name of a new class and a reference to the currently loaded AST. The method returns a *String* name that uses the base *String* passed concatenated with some random characters. The value of this method is that the returned class name is guaranteed to be unique in the model.

V. TEST RESULTS

5.1 Testing Approach

Each of the transforms was tested using various AWL input files. A transform was accepted as conforming to the requirements if both the *applicable* and *execute* methods of the transform worked as specified. In order for the *applicable* method to work as specified, it must detect any parts of the ASTs parsed from the AWL files that violate the rules set forth for the input the transform requires. In the case of *execute*, it will be considered working correctly if given valid input, it is able to transform the AST parsed from the AWL to support the relevant design pattern. In order to test these methods, we used equivalence classes. In the case of *applicable*, we could not separate the equivalence classes simply by whether it was valid input or not, due to the fact that any invalid input immediately cause the method to return *false*. In this case, we had one equivalence class for each pre-condition *applicable* was to check. In the case of each *execute* method, we had only one equivalence class: a valid input case. For both *execute* and *applicable,* we also always pass the entire AST as input, but we will omit this in future sections for the sake of brevity. Following this strategy, each transform test section is broken up into three parts: an explanation of the AWL file the transform will act on, a description of what was tested to ensure *applicable* was working properly, and an explanation showing the resultant AWL produced by the transform's *execute* method.

While we note that generating Java code from the result of transforming an AST has merit, this thesis did not create the Java code generator. In order to generate the Java code, we use the pre-existing Java code generator present in AWSOME. Noting that the

code generator had two limitations detrimental to our thesis, we updated the generator to correctly translate static attributes and to form correct syntax when the *new* operator is used to instantiate an object. After generating Java code from our AWL, we verified by visual inspection that the code was correct, that is to say that it contained no syntax errors or incorrect translation of the AST. Due to the limited relevance of the code to our thesis and for the sake of brevity, the results of such transforms are included in Appendix A.

5.2 Test Harness (Gump Panel)

The current layout of the AWSOME tool is that of an interactive GUI containing many buttons that a user may click to attempt applying a particular transform to an AST that has been loaded into the system. A new approach has been taken toward the GUI in the form of Java's tabbed panes. A tabbed pane is simply a component that allows buttons and other components to be added to it, but the luxury lies in the fact that many panes can be created and the user can click between them to only show one pane at a time. With each pane containing its own relevant components, the tool can maintain an organized layout and avoid the problem of a large number of buttons cluttering the screen.

Following the tabbed pane approach, this thesis introduces a new Java class entitled "GumpPanel.java," which we use as our test harness. A visual of this component is displayed in Figure 5.1. In order to facilitate modifications of the AST currently loaded into the system, the *GumpPanel* contains a reference to an AST that it is free to modify. The very top of the panel allows a user to browse the classes and methods in the loaded AST. When a class is selected, the middle left text field will display an outline of the entire class while the right text field will show the AWL for the class. The buttons in between the text fields and top browser are transforms related to previous work and are

not used in testing. The pane is also responsible for creating and showing buttons for each of the design pattern transforms described in this thesis, shown in the bottom middle. When a particular button for a transform is clicked by a user, the application is able to interactively query the user for any relevant information needed to complete the transform as well as deliver to the user all output. Using the user specified information and the reference to an AST, the pane is able to check if the transform underlying the clicked button can be applied. If the transform can be applied, the pane instantiates the relevant transform class and executes the transformation on the model. Once the transform is completed, the pane notifies the other GUI components that the model has changed so that they can be updated to reflect the current AST.

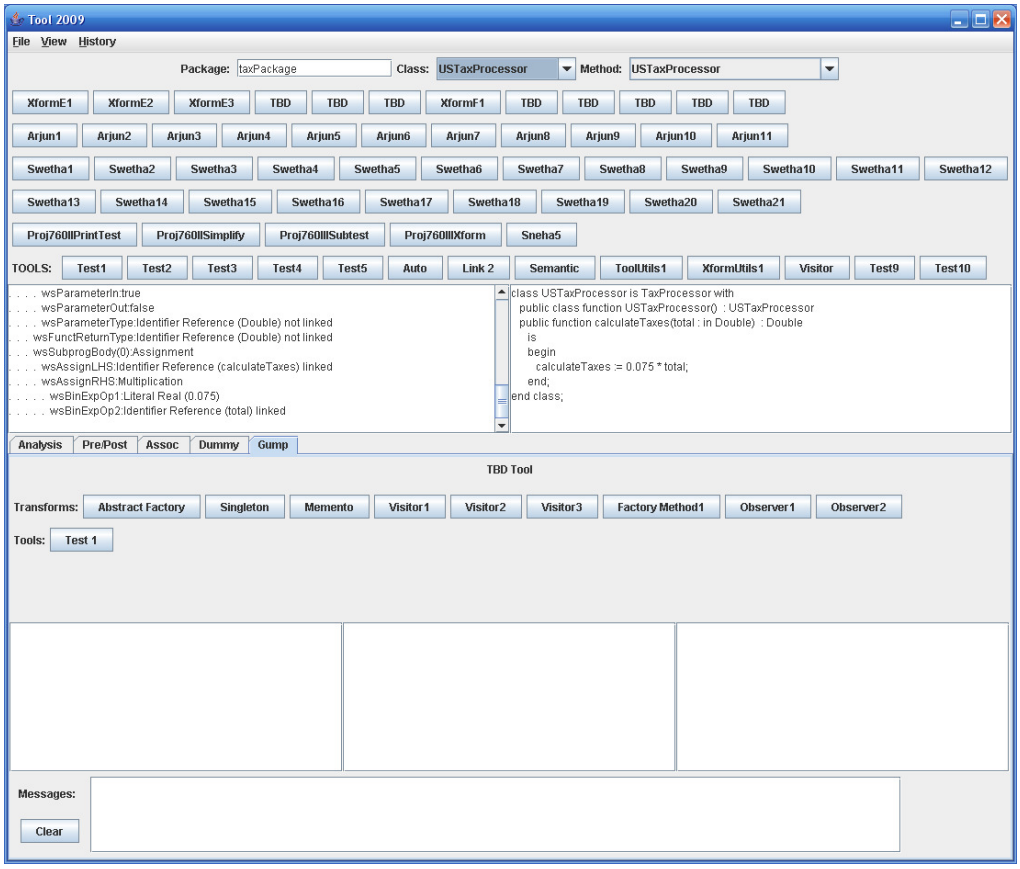

Figure 5.1: *GumpPanel* user interface.

5.3 Singleton Transform (XformSing1)

5.3.1 Test Case

The AWL file used to test the transform that updates a class to support the singleton pattern is shown in Figure 5.2. While we note that this model is seemingly complex, it is only made that way to address all modifications execute makes to the model. We first note that the type *double* has been declared; it may be thought of as a typical double point floating precision number common to programming languages such as Java. Next, we observe that the *CanadaTaxProcessor* class contains a default constructor, an attribute of our defined type *double*, and an additional class function. In order to satisfy the applicable method of the transform, the class does not contain an attribute named *singleton* or a method named *getInstance*.

```
package simpTax is

type double is digits 32 range -1.0e-1000 .. 1.0e+1000;

    class CanadaTaxProcessor is

       private rate : double;

     public class function CanadaTaxProcessor() : CanadaTaxProcessor
is
          begin
            rate := 0.05;
          end;

     public function calculateTaxes(total : in double) : double is
          begin
            calculateTaxes := rate * total;
     end;
    end class;

... other declarations used later omitted ...
```

Figure 5.2: Input AWL file to be used with singleton transform.

5.3.2 Applicable

To verify that the *applicable* method was working as intended, the following conditions were tested. To begin, we tried passing input that was not a class, to verify

that the transform only takes a class as an input. Next, we tried passing three classes to be transformed into a singleton, one of which had a *getInstance* method, another which had an attribute named *singleton*, and a final one that did not contain a default constructor. As expected, the *applicable* method did not allow any such classes to have the transform applied.

5.3.3 Execute

To test the *execute* method of the transform, we passed it the *CanadaTaxProcessor* class. The effects of the transform on the *CanadaTaxProcessor* class specified in AWL can be seen in Figure 5.3. We note that a private attribute of type *CanadaTaxProcessor* has been added to the class and the constructor has been converted to private (while we note the attribute is not tagged with a static keyword, it is stored as static internally in the system, and the code generator will generate the static keyword in Java). In addition, a new function has been added entitled *getInstance* that is responsible for instantiating the private *singleton* attribute and returning it to allow outside users to get an instance of the class. Furthermore, we see that the *getInstance* method is static by the use of the *class* keyword in is definition. Since this is exactly what is expected in a singleton class, we assert that this transform is working properly.

```
class CanadaTaxProcessor is
  private rate : double;
  private singleton : CanadaTaxProcessor;
  private class function CanadaTaxProcessor()  : CanadaTaxProcessor
    is begin
      rate := 0.05;
    end;
  public function calculateTaxes(total : in double)  : double
    is begin
      calculateTaxes := rate * total;
    end;
  public class function getInstance()  : CanadaTaxProcessor
    is begin
      if singleton = null then
        singleton := new CanadaTaxProcessor();
      else end if;
      getInstance := singleton;
    end;
end class;
```

Figure 5.3: The updated *CanadaTaxProcessor* class in AWL.

5.4 Singleton Usage Transform (XformSing2)

5.4.1 Test Case

The transform to test whether or not all classes in the AST are updated to utilize the newly created singleton uses the same AWL presented in Figure 5.2 with some modifications (these changes are shown in Figure 5.4). In this case, the emphasis is placed on the *CanadaTaxUser* class as it was created to use *CanadaTaxProcessor* objects. We see that the class utilizes a private attribute of type *CanadaTaxProcessor* which is assigned new instances in the various functions within the class. Of important note is the *testTaxer* function: while it does not have a seemingly useful effect, it contains several assignments where the private attribute is assigned a newly created *CanadaTaxProcessor* object. In order to test the thoroughness of the *execute* method of this transform, some of these assignments are nested in *if* statements. We note that *loops* are omitted in the test, but our implementation treats *ifs* and *loops* similarly so results would be the same. In

addition, we make use of another function, *testTax*, which also allocates a new *CanadaTaxProcessor* and assigns it to a local function variable. This particular method serves the function of ensuring that the *execute* method checks all functions for usage updates.

```
... previous code from Figure 5.2 ...
class CanadaTaxUser is

        private theTaxer : CanadaTaxProcessor;

      public class function CanadaTaxUser() : CanadaTaxUser is
          begin
      end;

      public procedure testTaxer() is
          rate : double;
          begin
            theTaxer := new CanadaTaxProcessor;
            theTaxer.calculateTaxes(20.0);
            rate := 1;
            if ( 5 > 6 ) then
                 theTaxer := new CanadaTaxProcessor;
            else
                 if ( 6 < 5 ) then
                     theTaxer := new CanadaTaxProcessor;
                  end if;
            end if;

            while ( rate < 5 ) do
                 if ( rate = 3 ) then
                     theTaxer := new CanadaTaxProcessor;
                 else
                     rate := rate + 1;
                 end if;
            end do;
      end;

        public function testTax(): double is
            theTaxer : CanadaTaxProcessor;
          begin
            theTaxer := new CanadaTaxProcessor;
            testTax := .05;
          end;
    end                                                    class;
end package;
```

Figure 5.4: Input AWL file to be used with singleton usage transform.

5.4.2 Applicable

In order to ensure that the *applicable* method was working as intended, we tried varying

inputs that the *applicable* method should report are invalid. To begin, we attempted passing input which was not a class. The *applicable* method returned *false* as expected since is requires a class as input. Next, we tested the method several times using various classes including one without a *getInstance* method, one without the *singleton* attribute, and one without a default constructor. Essentially, we passed classes that had none or some of the components of the singleton pattern applied, which the *applicable* method returned *false* for in all cases. Since this method correctly responded to all invalid inputs, it satisfies the requirements we imposed on the method.

5.4.3 Execute

In order to test the *execute* method, we first applied the singleton transform to the original AST parsed in from the AWL in Figure 5.4. Doing so created our initial AST to be used in conjunction with this transform. We then passed the *execute* method of the singleton usage transform the *CanadaTaxProcessor* class (updated to support singleton). The results of applying the transform can be seen in Figure 5.5. In this Figure, we show only the *CanadaTaxUser* class to enhance clarity. As expected, all lines that included the statement "theTaxer = new CanadaTaxProcessor();" have been replaced with the line "theTaxer = CanadaTaxProcessor.getInstance();". As we would expect, even the nested *if* statements that contained such statements have had the usage updated. Since the model has been updated precisely as we expected to make use of the singleton pattern, the usage transform adheres to the requirements we set forth for it.

```
class CanadaTaxUser is
  private theTaxer : CanadaTaxProcessor;
  public class function CanadaTaxUser()  : CanadaTaxUser

 public procedure testTaxer()
    is
      rate : double;
    begin
      theTaxer := CanadaTaxProcessor.getInstance();
      theTaxer.calculateTaxes(20.0);
      rate := 1;
      if 5 > 6 then
        theTaxer := CanadaTaxProcessor.getInstance();
      else
        if 6 < 5 then
          theTaxer := CanadaTaxProcessor.getInstance();
        else
        end if;
      end if;
      while rate < 5 do
        if rate = 3 then
          theTaxer := CanadaTaxProcessor.getInstance();
        else
          rate := rate + 1;
        end if;
      end do;
    end;

  public function testTax()  : double
    is
      theTaxer : CanadaTaxProcessor;
    begin
      theTaxer := CanadaTaxProcessor.getInstance();
      testTax := 0.05;
    end;
end class;
```

Figure 5.5: The AWL for the updated *CanadaTaxUser* class.

5.5 Abstract Factory Transform (XformAF1)

5.5.1 Test Case

The AWL file used for the abstract factory transform test can be seen in figures 5.6a-c. We begin by examining the first two classes declared: *TaxProcessor* and *SalaryCalc*.

Both of these classes are abstract and each contains one abstract function that must be implemented by any subclasses. Following the two classes are three concrete classes, *EuropeTaxProcessor*, *CanadaTaxProcessor*, and *USTaxProcessor*, which inherit from the *TaxProcessor* class and implement the abstract *calculateTaxes* function. This method simply takes in some total and returns the result of multiplying it by some percent. In addition, all three of these classes implement a default constructor. Following these three classes are three other classes, *EuropeSalarayCalc*, *CanadaSalaryCalc,* and *USSalaryCalc*, which inherit from the abstract *SalaryCalc* class and implement the *calcSalary* function. Again, the methods just do some arbitrary calculation and return the result. Furthermore, these classes also include default constructors.

```
package taxPackage is
    type double is digits 10 range -1.0e-20 .. 1.0e+20;
    class TaxProcessor is abstract
        public abstract function calculateTaxes(total : in double) :
double
    end class;

    class SalaryCalc is abstract
        public abstract function calcSalary(hours : in double) :
double
    end class;

    class EuropeTaxProcessor is TaxProcessor with
        public class function EuropeTaxProcessor() :
EuropeTaxProcessor is
          begin
            EuropeTaxProcess := new EuropeTaxProcessor;
          end;

        public function calculateTaxes(total : in double) : double
is
          begin
            calculateTaxes := 0.15 * total;
          end;
    end class;
```

… continued in Figure 6b …

Figure 5.6a: Beginning of input AWL file to be used with abstract factory transform.

```
... continued from figure 5.6 a ...
    class CanadaTaxProcessor is TaxProcessor with
        public class function CanadaTaxProcessor() :
CanadaTaxProcessor is
          begin
            CanadaTaxProcess := new CanadaTaxProcessor;
          end;

        public function calculateTaxes(total : in double) : double is
          begin
            calculateTaxes := 0.05 * total;
          end;
    end class;

       class USTaxProcessor is TaxProcessor with
        public class function USTaxProcessor() : USTaxProcessor is
          begin
            USTaxProcess := new USTaxProcessor;
          end;

    public function calculateTaxes(total : in double) : double is
          begin
            calculateTaxes := 0.075 * total;
          end;
    end class;

    class EuropeSalaryCalc is SalaryCalc with
       public class function EuropeSalaryCalc() : EuropeSalaryCalc is
          begin
            EuropeSalaryCalc := new EuropeSalaryCalc;
          end;

       public function calculateSalary(hours : in double) : double is
          begin
            calculateSalary := 7.25 * hours;
          end;
    end class;

    class CanadaSalaryCalc is SalaryCalc with
       public class function CanadaSalaryCalc() : CanadaSalaryCalc is
          begin
            CanadaSalaryCalc := new CanadaSalaryCalc;
          end;


       public function calculateSalary(hours : in double) : double is
          begin
            calculateSalary := 2.25 * hours;
          end;
    end class;

.. continued in Figure 5.8c …
```

Figure 5.6b: Continuation of input AWL file to be used with abstract factory transform.

```
... continued from Figure 5.8b ...


    class USSalaryCalc is SalaryCalc with
        public class function USSalaryCalc() : USSalaryCalc is begin
            USSalaryCalc := new USSalaryCalc; end;

        public function calculateSalary(hours : in double) : double
is
          begin
            calculateSalary := 20.00 * hours;
          end;
    end class;
end package;
```

Figure 5.6c: Continuation of input AWL file to be used with abstract factory transform.

5.5.2 Applicable

In order to test the *applicable* method, we attempted to pass it various inputs that violated the constraints imposed. We began by passing it "themes" of objects that did not have the same number of classes for each "theme" as well as classes that did not adhere to the strict parent/child class hierarchy demanded by the pattern. Next we tried passing it classes that did not contain a default constructor. In all of these cases the *applicable* method reported a violation and did not allow the pattern to be executed.

5.5.3 Execute

In order to test the abstract factory transform, we passed the *execute* method the abstract super classes *Taxprocessor* and *SalaryCalc* in addition to all the concrete subclasses of the two abstract classes. The resultant AWL file from the transform can be seen in figures 5.7a and 5.7b. For the sake of brevity, only the changes made to the AWL file by the transform are shown. We see that an abstract super class named *AbstractFactoryW* has been created with the abstract methods *createTaxProcessor* and *createSalaryCalc* which must be implemented by child classes. Furthermore, we see that a static method named *getFactory* has been created that takes an integer argument to

decide which concrete factory it should instantiate and return to the caller. Each of these concrete factories corresponds to the class declarations of *ConcreteFactoryS*, *ConcreteFactoryLGN*, or *ConcreteFactoryHMOXA*. These names were randomly generated to ensure uniqueness in the model and correspond to a "theme" of *TaxProcessor*/*SalaryCalc* objects. Each class implements the two abstract methods, with the difference being that the first class returns an object of the *Europe* type, the second conforms to the *Canada* type, while third returns objects that are the *US* type. Since the AWL file has been updated to have the form as set forth for a model transformed with the abstract factory pattern, this transform adheres to the requirements.

```
... previous declarations ...
  class ConcreteFactoryS is AbstractFactoryW with
      public function createTaxProcessor()  : TaxProcessor
         is
         begin
            createTaxProcessor := new EuropeTaxProcessor;
         end;
      public function createSalaryCalc()  : SalaryCalc
         is
         begin
            createSalaryCalc := new EuropeSalaryCalc;
         end;
   end class;
   class ConcreteFactoryLGN is AbstractFactoryW with
      public function createTaxProcessor()  : TaxProcessor
         is
         begin
            createTaxProcessor := new CanadaTaxProcessor;
         end;
      public function createSalaryCalc()  : SalaryCalc
         is
         begin
            createSalaryCalc := new CanadaSalaryCalc;
         end;
   end class;
... continued in Figure 5.9b ...
```

Figure 5.7a: The beginning of updated AWL file after applying abstract factory transform.

```
... continued from Figure 5.9a ...
class ConcreteFactoryHMOXA is AbstractFactoryW with
      public function createTaxProcessor()  : TaxProcessor
         is
         begin
            createTaxProcessor := new USTaxProcessor;
         end;
      public function createSalaryCalc()  : SalaryCalc
         is
         begin
            createSalaryCalc := new USSalaryCalc;
         end;
   end class;

   class AbstractFactoryW is abstract
      public abstract function createTaxProcessor()  : TaxProcessor
      public abstract function createSalaryCalc()  : SalaryCalc
      public class function getFactory(factoryChoice : in Integer)  :
AbstractFactoryW
         is
         begin
            if factoryChoice = 1 then
               getFactory := new ConcreteFactoryS;
            else
            end if;
            if factoryChoice = 2 then
               getFactory := new ConcreteFactoryLGN;
            else
            end if;
            if factoryChoice = 3 then
               getFactory := new ConcreteFactoryHMOXA;
            else
            end if;
         end;
   end class;
end package;
```

Figure 5.7b: The end of updated AWL file after applying abstract factory transform.

5.6 Abstract Factory Usage Transform

5.6.1 Test Case

The test case created for the usage transform test builds off of the AWL file in Figure 5.6. To make these changes clearer, Figure 5.8 shows only the newly added classes in the AWL file. To this AWL file we add two new classes: *User* and *User2*. Each of these classes contains two attributes defined as one of the concrete class types from the previous AWL model. Each class has a default constructor as well as a single method in which we instantiate the appropriate class for each attribute and call a relevant class

function. This setup is chosen to illustrate that the transform works for multiple types in multiple classes and functions.

```
... other declarations ...
class User is
        private taxer : USTaxProcessor;
        private calcer : USSalaryCalc;
        public class function User() : User is
          begin  end;

        public procedure tester() is
          begin
            taxer := new USTaxProcessor;
            calcer := new USSalaryCalc;
            if ( 5 > 6 ) then
                 calcer := new USSalaryCalc;
            else
                   if ( 6 < 5 ) then
                     taxer := new USTaxProcessor;
                  end if;
            end if;
            taxer.calculateTaxes(25);
            calcer.calcSalary(40);
          end;    end class;
    class User2 is
        private taxer : CanadaTaxProcessor;
        private calcer : CanadaSalaryCalc;
        public class function User2() : User2 is
          begin  end;

        public procedure tester() is
          begin
            taxer := new CanadaTaxProcessor;
                 calcer := new CanadaSalaryCalc;
            if ( 5 > 6 ) then
                 calcer := new CanadaSalaryCalc;
            else
                   if ( 6 < 5 ) then
                     taxer := new CanadaTaxProcessor;
                  end if;
            end if;
            taxer.calculateTaxes(25);
            calcer.calcSalary(40);
        end;    end class;
```

Figure 5.8: Input AWL file to be used with abstract factory usage transform.

5.6.2 Applicable

In order to test the *applicable* method, we were forced to pass many different inputs to the method. To begin, we attempted passing it input that did not have the required list of

lists format with an abstract parent class being the first element of each list and its subclasses being the subsequent elements. Next, we attempted to pass it the previous input without also passing it the newly created abstract factory. Finally, we checked one of the most important constraints by passing the transform input classes that used the concrete children of the abstract classes, but called methods in the concrete classes that did not exist in the abstract parent class. In all of these cases, the *applicable* method caught the errors in input and reported that the transform could not be applied.

5.6.3 Execute

To test the *execute* method, we passed it the input classes *SalaryCalc* and *TaxProcessor* (as well as all of their concrete subclasses) and the abstract factory class created by applying the previous transformation. The result of applying the transform is shown in Figure 5.9, though we show only the changes to the *User* and *User2* classes. As we can see, the *User* class has had a private attribute of type *AbstractFactoryW*, aptly named *theAbstractFactoryW*, added to it to store the abstract factory and the old attributes' types have been changed to the abstract super class type. In the *tester* procedure we call the abstract factory's static *getFactory* method and pass it 3 to retrieve the factory associated with the "US theme." Next, we notice that all places that previously instantiated concrete subclasses have been replaced with a call to *theAbstractFactoryW*'s *create* method, whether it be for a *TaxProcessor* or a *SalaryCalc*. These calls will return the appropriate "US theme" for the objects since the factory has been set to the concrete factory that produces that "theme." The *User2* class has had similar changes, except that its *getFactory* call passes 2 to retrieve the "Canada theme." Since this AWL file has the form required to implement the abstract factory usage

transform, the *execute* method satisfies the proposed requirements.

```
... other declarations ...
class User is
      private taxer : TaxProcessor;
      private calcer : SalaryCalc;
      private theAbstractFactoryW : AbstractFactoryW;
      public class function User()  : User
         is begin end;

      public procedure tester() is begin
            theAbstractFactoryW := AbstractFactoryW.getFactory(3);
            taxer := theAbstractFactoryW.createTaxProcessor();
            calcer := theAbstractFactoryW.createSalaryCalc();
            if 5 > 6 then
               calcer := theAbstractFactoryW.createSalaryCalc();
            else
               if 6 < 5 then
                  taxer := theAbstractFactoryW.createTaxProcessor();
               else end if;
            end if;
            taxer.calculateTaxes(25); calcer.calcSalary(40);
         end;     end class;

class User2 is
      private taxer : TaxProcessor;
      private calcer : SalaryCalc;
      private theAbstractFactoryW : AbstractFactoryW;
      public class function User2()  : User2
         is begin end;

      public procedure tester() is begin
            theAbstractFactoryZD := AbstractFactoryW.getFactory(2);
            taxer := theAbstractFactoryW.createTaxProcessor();
            calcer := theAbstractFactoryW.createSalaryCalc();
            if 5 > 6 then
               calcer := theAbstractFactoryW.createSalaryCalc();
            else
               if 6 < 5 then
                taxer := theAbstractFactoryW.createTaxProcessor();
               else end if;
            end if;
            taxer.calculateTaxes(25); calcer.calcSalary(40);
         end;     end class;
```

Figure 5.9: AWL generated from executing abstract factory usage transform.

5.7 Factory Method Transform (XformFactM1)

5.7.1 Test Case

In order to test the factory method transform, an AWL file was created that is precisely the same as the one presented in Figure 5.6. For this reason, we omit showing it again.

The reason we use the same model is that the factory method transform is also related to super class/multiple subclass hierarchies.

5.7.2 Applicable

In order to test the *applicable* method, we passed a variety of inputs that specifically violated a rule the method was looking to enforce. To begin, we attempted to pass input that was not a list of class names in the model as *Strings*. We also attempted to pass *applicable* a list of classes that did not inherit from a common super class. Next we tried to pass classes that had a super class that was not already in the model. After that test, we sent the transform input that had already been transformed to support the pattern. Finally, we attempted to use input classes that did not contain default constructors. In all the listed cases, the *applicable* method caught the error and reported it, without allowing the *execute* method to be performed.

5.7.3 Execute

To test the *execute* method, we passed it the input classes *USTaxProcessor*, *CanadaTaxProcessor*, and *EuropeTaxProcessor*. The result of applying the transform to the specified input can be seen in Figure 5.10. Note that only the changes in the AWL are shown to enhance clarity. We note that a new class has been added to the model named *TaxProcessorFactory* that is responsible for providing the right type of concrete subclass of the *TaxProcessor* class. In this case, we have three private *boolean* attributes in the class that are used to indicate what type of concrete class should be returned by the *getInstance* method. The *getInstance* method simply checks each flag and instantiates and returns the appropriate type if the matching *boolean* attribute was *true*. The remaining methods are *useEuropeTaxProcessor*, *useCanadaTaxProcessor*, and

useUSTaxProcessor which simply set the relevant b*oolean* attribute to *true* and all others to *false*. As this entire class provides the functionality given by applying the factory method design pattern, the *execute* method adheres to the requirements previously set forth.

```
... other declarations ...
class TaxProcessorFactory is
      private EuropeTaxProcessorFlag : boolean;
      private CanadaTaxProcessorFlag : boolean;
      private USTaxProcessorFlag : boolean;
      public class function TaxProcessorFactory()  :
TaxProcessorFactory
      public function getInstance()  : TaxProcessor
         is
            tmp : TaxProcessor;
         begin
            if EuropeTaxProcessorFlag = true then
               tmp := new EuropeTaxProcessor;
            else end if;
            if CanadaTaxProcessorFlag = true then
               tmp := new CanadaTaxProcessor;
            else end if;
            if USTaxProcessorFlag = true then
               tmp := new USTaxProcessor;
            else end if;
            getInstance := tmp;
         end;
      public procedure useEuropeTaxProcessor()
         is begin
            EuropeTaxProcessorFlag := true;
            EuropeTaxProcessorFlag := false;
            EuropeTaxProcessorFlag := false;
         end;
      public procedure useCanadaTaxProcessor()
         is begin
            CanadaTaxProcessorFlag := false;
            CanadaTaxProcessorFlag := true;
            CanadaTaxProcessorFlag := false;
         end;
      public procedure useUSTaxProcessor()
         is begin
            USTaxProcessorFlag := false;
            USTaxProcessorFlag := false;
            USTaxProcessorFlag := true;
         end;
   end class;
```

Figure 5.10: AWL generated from executing the factory method transform.

5.8 Factory Method Usage Transform (XformFactM2)

5.8.1 Test Case

In order to test the usage transform for the factory method, we make use of the same model from Figure 5.6, with one addition. In this case, our addition is a whole new class entitled *CanadaTaxUser*. The AWL used as input for this transform is shown in Figure 5.11, with the similarities to the previous AWL file omitted. We see that the class has a private attribute of type *CanadaTaxProcessor*. In addition, there are two methods, the first, *testTaxer*, contains several *if* statements as well as *while* loops that serve only to allow us to nest instantiations of the private attribute. The second method, *testTax*, contains a private local variable that is used to demonstrate that we can also update usage for both local variables and multiple methods.

```
... other declarations ...
class CanadaTaxUser is

        private theTaxer : CanadaTaxProcessor;
     public class function CanadaTaxUser() : CanadaTaxUser is
          begin end;

     public procedure testTaxer() is
          rate : double; begin
            theTaxer := new CanadaTaxProcessor;
            theTaxer.calculateTaxes(20.0);
            rate := 1;
            if ( 5 > 6 ) then
                 theTaxer := new CanadaTaxProcessor;
            else if ( 6 < 5 ) then
                     theTaxer := new CanadaTaxProcessor;
                  end if;
            end if;
            while ( rate < 5 ) do
            if ( rate = 3 ) then
                 theTaxer := new CanadaTaxProcessor;
                 else rate := rate + 1;
            end if;
        end do;
      end;

       public function testTax(): double is
            theTaxer2 : EuropeTaxProcessor; begin
            theTaxer2 := new EuropeTaxProcessor;
            testTax := 0.05;
     end;
    end class;
```

Figure 5.11: Input AWL file to be used with factory method usage transform.

5.8.2 Applicable

In following our previous strategy to test *applicable*, we simply pass various inputs to the method that violate the constraints it enforces. To begin, we attempt to pass input that is not a vector of class names in the model as *Strings*. Next we attempt to pass a list of classes that do not share a common super class. In following the super class idea, we pass a list of classes with a super class not present in the model. Next we tried to pass a model that had classes with attributes named "theSuperClassFactory" where SuperClass is the name of the list of classes' common super class. In addition, we tried to pass a model that had not been transformed by the factory method transform (it lacked a class named SuperClassFactory, where SuperClass is the common super class to the input classes). After that test, we tried to send an input model that used the concrete subclasses specified, but called methods that were not in their abstract parent classes. Finally, we tried passing classes that did not have default constructors as well. Since the *applicable* method correctly identified all these input errors and reported them, it passed the requirements specified for it.

5.8.3 Execute

In order to test the *execute* method, we passed it the input classes *USSalaryCalc*, *CanadaSalaryCalc*, and *EuropeSalaryCalc*. The result of applying the transform to said inputs can be seen in Figure 5.12. To enhance clarity, only the *CanadaTaxUser* class is shown, since it was the only class updated by the transform. The first note we make is that the private attribute has been transformed into the abstract super class type *TaxProcessor*. Furthermore, a new attribute has been added to store the *TaxProcessorFactory* that was created in the previous transform. We also see that the

constructor of the class has been updated to instantiate the new attribute. Moving on to the *testTaxer* method, we note that initially the function calls *theTaxProcessorFactory*'s *useCanadaTaxProcessor* method to correctly set up the factory object. It then replaces all previous instantiations that used the *new* operator with a method call: *theTaxProcessorFactory.getInstance()*. In the case of the second function, *testTax*, we see that a similar change has occurred, with the exception that we do not change the local variables type to the abstract super class. Since the AWL file matches the expected result specified by the requirements, the *execute* method correctly implements the usage transform.

```
... other declarations ...
class CanadaTaxUser is
      private theTaxer : TaxProcessor;
      private theTaxProcessorFactory : TaxProcessorFactory;
      public class function CanadaTaxUser()  : CanadaTaxUser
         is begin theTaxProcessorFactory := new TaxProcessorFactory;
         end;
      public procedure testTaxer()
         is rate : double; begin
            theTaxProcessorFactory.useCanadaTaxProcessor();
            theTaxer := theTaxProcessorFactory.getInstance();
            theTaxer.calculateTaxes(20.0);
            rate := 1;
            if 5 > 6 then
               theTaxer := theTaxProcessorFactory.getInstance();
            else
               if 6 < 5 then
                  theTaxer := theTaxProcessorFactory.getInstance();
               else
               end if;
            end if;
            while rate < 5 do
               if rate = 3 then
                  theTaxer := theTaxProcessorFactory.getInstance();
               else rate := rate + 1;
               end if;
            end do; end;
      public function testTax()  : double
         is theTaxer2 : EuropeTaxProcessor; begin
            theTaxProcessorFactory.useEuropeTaxProcessor();
            theTaxer2 := theTaxProcessorFactory.getInstance();
            testTax := 0.05; end;
   end class;
```

Figure 5.12: AWL generated from executing factory method usage transform.

5.9 Memento (XformMem1)

5.9.1 Test Case

In order to test the memento transform, we created an AWL file to serve as input as shown in Figure 5.13a-b. We note that this class does not do anything particularly useful, but rather it contains various types of attributes to test the functionality of the memento transform. First we note that two of the attributes, *rate* and *num*, are both of primitive types, to test that they can be copied correctly. Next we have *other* which is of type *otherClass*, which contains a method to copy an instance of itself. In addition, we have an attribute named *arr* which can be thought of as an array. Finally, we have an attribute that provides access to *arr* named *arrP*. We note that the class also contains methods to copy each of these attributes: *copyarr* and *copyarrP*. While these copy methods do not make "deep" copies, they are included to illustrate the point that the memento pattern transform will seek out copy methods for non-primitive attributes.

```
package simpTax is

type double is digits 32 range -1.0e-20.0 .. 1.0e+20.0;
type integer is range -100 .. 100;
type TArray is bag of otherClass;
type TPtr is access TArray;

class CanadaTaxProcessor is
        private rate : double;
        private other : otherClass;
        private num : integer;
        private arr : TArray;
        private arrP : TPtr;

... continued in figure 5.13b ...
```

Figure 5.13a: Beginning of input AWL file to be used with memento transform.

```
... continued from figure 5.13a ...
        public class function CanadaTaxProcessor() :
CanadaTaxProcessor
        is begin
            rate := 0.05;
          end;

        public function calculateTaxes(total : in double) : double is
           sample : double;
          begin
            calculateTaxes := rate * total;
          end;

      public function copyarr() : TArray is begin
            copyarr := arr;
          end;

      public function copyarrP() : TPtr is begin
            copyarrP := arrP;
          end;
    end class;

      class otherClass is
        public function copy() : otherClass is begin
            copy := new otherClass;
          end;
    end class;
end package;
```

Figure 5.13b: Continuation of input AWL file to be used with memento transform.

5.9.2 Applicable

In order to test whether or not the *applicable* method enforces the restrictions placed on input, we tried passing several different inputs to the method. To begin, we attempted to pass it a non-class argument and then attempted passing a class with no attributes. We then tried passing the function a class that already had a *setMemento* and *createMemento* function in it. Next we used an input AST that already had a class named "ClassMemento" where Class was the name of the input class we specified. Finally, we passed the method an input class that contained non primitive elements, but did not have a method defined of the form "copyName" where Name is the name of a non-primitive attribute. In all cases, the *applicable* method detected and reported an error on the expected input.

5.9.3 Execute

In testing the *execute* method, we passed it the input class *CanadaTaxProcessor*. The resultant AWL file from the transform can be seen in figures 5.14a-d. As expected, a new class has been created, *CanadaTaxProcessorMemento*, which has precisely the same attributes as the input class. In addition, this class has *get* and *set* methods for each of the attributes as well as a constructor that takes all the attributes as input. Next we examine the *CanadaTaxProcessor* class which has been updated with two new methods. The first method, *setMemento*, has been created that takes a *CanadaTaxProcessorMemento* as an argument and sets the internal attributes of the *CanadaTaxProcessor* to the values returned by calling each *get* method of the argument. Next, we see that the other function, *getMemento*, has been created which begins by declaring local variables for all the attributes in the class as well as a *CanadaTaxProcessorMemento*. Next, the function makes a copy of each attribute either by assignment (for primitive types) or calls to the attributes relevant copy method (in the case of objects and containers). The *CanadaTaxProcessorMemento* object is then instantiated and each of its set methods are called, passing the previously mentioned local variables to each one. Finally, the completed object is returned by the function. Since this AWL appears to satisfy the requirements outlined for the memento transform, the *execute* method of this transform passes the acceptance test.

```
package simpTax is
   type double is digits 32 range -1.0e-20.0 .. 1.0e+20.0;
   type int is range -100 .. 100;
   type TArray is bag of otherClass;
   type TPtr is access TArray;
   class CanadaTaxProcessor is
      private rate : double;
      private other : otherClass;
      private num : int;
      private arr : TArray;
      private arrP : TPtr;
      public class function CanadaTaxProcessor()  :
CanadaTaxProcessor
         is begin rate := 0.05; end;
      public function calculateTaxes(total : in double)  : double is
            sample : double;
         begin
            calculateTaxes := rate * total;
         end;
      public function copyarr()  : TArray is begin
            copyarr := arr;
         end;
      public function copyarrP()  : TPtr is begin
            copyarrP := arrP;
         end;

      public procedure setMemento(theCanadaTaxProcessorMemento :
CanadaTaxProcessorMemento)
         is
         begin
            rate := theCanadaTaxProcessorMemento.getrate();
            other := theCanadaTaxProcessorMemento.getother();
            num := theCanadaTaxProcessorMemento.getnum();
            arr := theCanadaTaxProcessorMemento.getarr();
            arrP := theCanadaTaxProcessorMemento.getarrP();
         end;

... continued in Figure 5.20b ...
```

Figure 5.14a: AWL generated from executing the memento transform.

```
... continued from Figure 5.20a ...

      public function getMemento()  : CanadaTaxProcessorMemento
         is
            localMem : CanadaTaxProcessorMemento;
            localrate : double;
            localother : otherClass;
            localnum : int;
            localarr : TArray;
            localarrP : TPtr;
         begin
            localMem := new CanadaTaxProcessorMemento;
            localrate := rate;
            localother := other.copy();
            localnum := num;
            localarr := copyarr();
            localarrP := copyarrP();
            localmem.setrate(localrate);
            localmem.setother(localother);
            localmem.setnum(localnum);
            localmem.setarr(localarr);
            localmem.setarrP(localarrP);
            getMemento := localMemento;
         end;
   end class;
class otherClass is
      public function copy()  : otherClass
         is
         begin
            copy := new otherClass;
         end;
   end class;

class CanadaTaxProcessorMemento is
      private rate : double;
      private other : otherClass;
      private num : int;
      private arr : TArray;
      private arrP : TPtr;
      public class function CanadaTaxProcessorMemento(rate1 : in
double, other1 : in otherClass, num1 : in int, arr1 : in TArray,
arrP1 : in TPtr)  :
         is
         begin
         rate := rate1;
            other := other1;
            num := num1;
            arr := arr1;
            arrP := arrP1;
         end;

... continued in Figure 5.20c ...
```

Figure 5.14b: Continued AWL generated from executing the memento transform.

```
… continued from Figure 5.20b …


      public function getrate()  : double
         is
         begin
            getrate := rate;
         end;
      public function getother()  : otherClass
         is
         begin
            getother := other;
         end;
      public function getnum()  : int
         is
         begin
            getnum := num;
         end;
      public function getarr()  : TArray
         is
         begin
            getarr := arr;
         end;
      public function getarrP()  : TPtr
         is
         begin
            getarrP := arrP;
         end;
      public procedure setrate(therate : in double)
         is
         begin
            rate := therate;
         end;
      public procedure setother(theother : in otherClass)
         is
         begin
            other := theother;
         end;
      public procedure setnum(thenum : in int)
         is
         begin
            num := thenum;
         end;
```

… continued in Figure 5.20d …

Figure 5.14c: Continued AWL generated from executing the memento transform.

```
... continued from Figure 5.20c ...

      public procedure setarr(thearr : in TArray)
         is
         begin
            arr := thearr;
         end;
      public procedure setarrP(thearrP : in TPtr)
         is
         begin
            arrP := thearrP;
         end;
   end class;
end package;
```

Figure 5.14d: Continued AWL generated from executing the memento transform.

5.10 Observer Transform (XformObs1)

5.10.1 Test Case

In order to develop a test case for this transform, we have no real requirement other than an AWL file that can be parsed into an AST. In choosing such a model, we again return to our *CanadaTaxProcessor* favorite as shown in Figure 5.15. We note that the AWL is very simple and the *CanadaTaxProcessor* class is included so that the file is not blank, but it certainly is not necessary.

```
package simpTax is

type double is digits 32 range -1.0e-100 .. 1.0e+100;
type int is range -100 .. 100;

    class CanadaTaxProcessor is

        private rate : double;
        private num : int;

        public class function CanadaTaxProcessor() :
CanadaTaxProcessor is begin
            rate := 0.05;
          end;

        public function calculateTaxes(total : in double) : double is
          sample : double; begin
            calculateTaxes := rate * total;
          end;

    end class;
end package;
```

Figure 5.15: Input AWL file to be used with observer transform.

5.10.2 Applicable

Keeping with the standard method of testing the *applicable* method, we attempted feeding it various invalid inputs it was designed to detect. To begin, we attempted passing it an AST that already had a class named *Observable* in it. Next, we use an AST that contained an *Observer* class as input to the transform. In both cases, the method caught the illegal input and reported an error. For this reason, the *applicable* method is performing as specified.

5.10.3 Execute

In order to test the *execute* method, we did not need to send it any actual input. The results of applying the transform can be seen in Figure 5.16. We first note that two new classes have been added to the model. The first class is the abstract *Observer* class that provides one abstract method: *update*. Since this class is abstract, it is required that any class that wants to be an *Observer* must inherit from it and implement that method. The next class, *Observable*, is abstract and contains three methods that a subclass must implement. The first method is *attach*, which takes an *Observer* as input and should be implemented to add the input to some internal class storage. The second, *detach*, also takes an *Observer* as input and should be used to remove the input from the internal class storage. Finally, the *notify* method is included in the class and should be used to iterate through the internal storage of *Observers* and call each one's *update* method. Since the produced AWL adheres to the requirements specified for the observer transform, the *execute* method operates precisely as expected.

```
package simpTax is
   type double is digits 32 range -1.0e-100 .. 1.0e+100;
   type int is range -100 .. 100;
   class CanadaTaxProcessor is
      private rate : double;
      private num : int;

      public class function CanadaTaxProcessor()  :
CanadaTaxProcessor
         is
         begin
            rate := 0.05;
         end;

      public function calculateTaxes(total : in double)  : double
         is
            sample : double;
         begin
            calculateTaxes := rate * total;
         end;
   end class;

   class Observer is abstract
      public abstract procedure update()
   end class;
   class Observable is abstract
      public procedure attach(o : in Observer)
      public procedure detach(o : in Observer)
      public procedure notify()
   end class;
end package;
```

Figure 5.16: AWL generated from executing the observer transform.

5.11 Observer Add Class Transform (XformObs2)

5.11.1 Test Case

In order to test the observer add class transform, we build on the results of applying the previous transform and use the AWL file outlined in Figure 5.16. We use this class for several reasons, the first of which is that the transform requires that the *Observer* and *Observable* classes already exist in the model. In addition, this class was chosen so that we had another class, *CanadaTaxProcessor* (initially shown in Figure 5.15), that we could use as input to the transform.

5.11.2 Applicable

In order to test the *applicable* method, we passed various illegal inputs to the method and noted the results. To begin, we attempted passing an AST that contained no class named *Observer* as well as an AST that contained a class named *Observer*, but the class lacked an *update* method. We then attempted passing the method an AST that contained no class named *Observable* as well as passing it an AST with a class name *Observable*, that lacked one or more of the methods *attach*, *detach*, or *notify*. Next we attempted passing it an input that was not a class to apply the transform to. Finally, we tried passing a class that already had a super class as well as a class that already contained an *update* method. In all the mentioned cases, the *applicable* method caught the invalid input and reported an error.

5.11.3 Execute

To test the *execute* method, we passed it the input class *CanadaTaxProcessor*. The results of the transform can be seen in Figure 5.17. As seen in the Figure, the only class to undergo any changes was the *CanadaTaxProcessor* class. We first note that it has been changed to subclass the *Observer* class which means it must implement the *update* method or be forced to be abstract as well. Fortunately, we see that it does indeed also implement the *update* method. Although the method has no body, it is not declared as abstract and thus is syntactically correct. Since this has in essence turned the *CanadaTaxProcessor* class into an *Observer*, the *execute* method for this transform is working correctly.

```
package simpTax is
   type double is digits 32 range 1.0e-100 .. 1.0e+100;
   type int is range -100 .. 100;

   class CanadaTaxProcessor is Observer with
      private rate : double;
      private num : int;

      public class function CanadaTaxProcessor()  :
CanadaTaxProcessor
         is
         begin
            rate := 0.05;
         end;

      public function calculateTaxes(total : in double)  : double
         is
            sample : double;
         begin
            calculateTaxes := rate * total;
         end;

      public procedure update()
   end class;

   class Observer is abstract
      public abstract procedure update()
   end class;

   class Observable is
      public procedure attach(o : in Observer)
      public procedure detach(o : in Observer)
      public procedure notify()
   end class;
end package;
```

Figure 5.17: AWL generated from executing the observer add class transform.

5.12 Visitor Add Super Class Transform (XformVis1)

5.12.1 Test Case

In order to test the visitor transform, we return to the *TaxProcessor* idea. However, since AWSOME does not support the idea of multiple inheritance, we instead strip out the notion of the parent classes and are left with the AWL in Figure 5.18. For the sake of brevity, we eliminate the "Canada theme" of classes. In using this model, our aim is to simply have several classes with existing methods that would want to allow an abstract parent class to be created.

```
package visPackage is
   type double is digits 10 range -1.0e-20 .. 1.0e+20;
    class EuropeTaxProcessor is
        public class function EuropeTaxProcessor() :
EuropeTaxProcessor is
          begin
            EuropeTaxProcess := new EuropeTaxProcessor; end;

        public function calculateTaxes(total : in double) : double is
          begin
            calculateTaxes := 0.15 * total;  end;
    end class;

    class USTaxProcessor is
        public class function USTaxProcessor() : USTaxProcessor is
begin
            USTaxProcess := new USTaxProcessor; end;

        public function calculateTaxes(total : in double) : double is
begin
            calculateTaxes := 0.075 * total; end;
    end class;

    class EuropeSalaryCalc is
        public class function EuropeSalaryCalc() : EuropeSalaryCalc
is
          begin
            EuropeSalaryCalc := new EuropeSalaryCalc; end;

        public function calculateSalary(hours : in double) : double
is begin
            calculateSalary := 7.25 * hours; end;
    end class;

    class USSalaryCalc is
        public class function USSalaryCalc() : USSalaryCalc is begin
            USSalaryCalc := new USSalaryCalc;  end;

        public function calculateSalary(hours : in double) : double
is begin
            calculateSalary := 20.00 * hours; end;
    end class;
end package;
```

Figure 5.18: Input AWL file to be used with visitor transform.

5.12.2 Applicable

In keeping with the common strategy of testing *applicable*, we attempt passing it input that it is designed to guard against. We began by sending the method a list of objects that were not classes. Next, we tried sending it a list of classes without also specifying the name for the new class to be created as well as a list of classes that were not in the model.

Our next input was a list of classes and a new class name, but the list of classes already inherited from some super class. Finally, we attempted to pass the method a new class name that was already present in the method. In all the cases presented, *applicable* detected and reported an error in the input.

5.12.3 Execute

In order to test the *execute* method, we the input classes *EuropeTaxProcessor*, *USTaxProcessor*, *EuropeSalaryCalc*, and *USSalaryCalc* as well as the new class name *Acceptor*. The results of apply the transform to the specified input can be seen in Figure 5.19. We note that at the very bottom of the Figure, the new class, *Acceptor*, has been defined as abstract with no attributes or methods. Furthermore, the previous four classes have all been updated to inherit from the *Acceptor* class (we only display *EuropeTaxProcessor* for the sake of brevity). These changes serve as evidence that the *execute* method is working as per its requirements.

```
package visPackage is
   type double is digits 10 range -1.0E-20 .. 1.0E20;

   class EuropeTaxProcessor is Acceptor with
      public class function EuropeTaxProcessor()  :
EuropeTaxProcessor
         is
         begin
            EuropeTaxProcess := new EuropeTaxProcessor;
         end;
      public function calculateTaxes(total : in double)  : double
         is
         begin
            calculateTaxes := 0.15 * total;
         end;
   end class;

... other classes ...

   class Acceptor is abstract
   end class;

end package;
```

Figure 5.19: AWL generated from executing the add super class transform.

5.13 Visitor Transform (XformVis3)

5.13.1 Test Case

The observant reader may note that we have skipped from XformVis1 to XformVis3. We note that it is simpler to apply XformVis3 and then apply XformVis2 to the result and thus we test them in that order. In order to test the visitor transform, we make use of the AWL file created in 5.19. To reduce clutter, we eliminated the *EuropeSalaryCalc* and *USSalaryCalc* classes noting that applying the pattern to only two classes is the same as applying it to four. The *Acceptor* class and the inheritance relationship is maintained, however, as the pattern requires that the target classes inherit from some common super class.

5.13.2 Applicable

In testing *applicable* we again pass various inputs to the method that it is was designed to guard against. We began by passing it a list that did not contain classes as well as passing it a list without a name for the new visitor class. We then attempted sending the method an AST that already had a class with the same name as the new visitor class we specified. Next, we sent the method a list of classes as input that did not all inherit from a common super class. Finally, we attempted sending it a list of classes that already had a method named *Accept* as well as a list of classes that had a parent class that already had an *Accept* method. In all of the tested cases, the *applicable* method reported an error in the input.

5.13.3 Execute

In order to test the *execute* method, we sent it the input classes *EuropeTaxProcessor* and *USTaxProcessor* and the new class named *TaxProcessor*. The results of executing

the transform on the specified input can be seen in Figure 5.20a-b. We first note that a new abstract class, *TaxProcessorVisitor*, has been created which declares two abstract methods: *VisitUsTaxProcessor* and *VisitEuropeTaxProcessor*. Both of these methods take as an argument their respective *TaxProcessor* type. Next, we note that the *Acceptor* class has been updated with a new abstract method, *Accept*, that takes as an argument the abstract *TaxProcessorVisitor* class. Since this method is abstract, the subclasses of *Acceptor* must implement them or be forced to be abstract as well. As we see, the *EuropeTaxProcessor* and *USTaxProcessor* classes have been updated to implement the *Accept* method. In each case, they simply call the *TaxProcessorVisitor*'s appropriate method and pass themselves (*this*) as an argument. Since the structure of this AWL file has precisely the structure previously described in the requirements, the *execute* method is considered to be functioning properly.

```
package visPackage is
   type double is digits 10 range -1.0E-20 .. 1.0E20;
   class EuropeTaxProcessor is Acceptor with
      public class function EuropeTaxProcessor()  :
EuropeTaxProcessor
         is  begin
            EuropeTaxProcess := new EuropeTaxProcessor;
         end;
      public function calculateTaxes(total : in double)  : double
         is  begin
            calculateTaxes := 0.15 * total;
         end;
      public procedure Accept(visitor : in TaxProcessorVisitor)
         is  begin
            visitor.VisitEuropeTaxProcessor(this);
         end;
   end class;
```

… continued in Figure 5.20b …

Figure 5.20a: Begginning of input AWL file from executing the visitor transform.

```
... continued from Figure 5.20a ...
   class USTaxProcessor is Acceptor with
      public class function USTaxProcessor()  : USTaxProcessor
         is begin
            USTaxProcess := new USTaxProcessor;
         end;
      public function calculateTaxes(total : in double)  : double
         is begin
            calculateTaxes := 0.075 * total;
         end;
      public procedure Accept(visitor : in TaxProcessorVisitor)
         is begin
            visitor.VisitUSTaxProcessor(this);
         end;
   end class;

   class Acceptor is abstract
      public abstract procedure Accept(visitor : in
TaxProcessorVisitor)
   end class;

   class TaxProcessorVisitor is abstract
      public abstract procedure VisitEuropeTaxProcessor(node : in
EuropeTaxProcessor)
      public abstract procedure VisitUSTaxProcessor(node : in
USTaxProcessor)
   end class;
end package;
```

Figure 5.20b: Continuation of AWL generated from executing the visitor transform.

5.14 Visitor Add Attribute Transform (XformVis2)

5.14.1 Test Case

Rather than use an entire new test case for the add attribute transform, we instead opt to continuing building on the AWL file displayed in Figure 5.20. This decision is fueled by the fact that the transform being tested expects that a Visitor transform would have been conducted prior. However, we add an additional class to the model as shown in Figure 5.21, with the intent to store extra information about some "secret tax."

```
... other declarations ...

   class InfoClass is
       public secretTax : double;

       public class function InfoClass() : InfoClass is begin
            secretTax := 0.05; end;
   end class;
```

Figure 5.21: Input AWL file to be used with add attribute transform.

5.14.2 Applicable

In our test of the *applicable* method, we chose to send several different inputs that violated the constraints the method was created to enforce. To begin, we attempted sending the method a list of arguments that were not *Strings*. Next, we attempted to send it a number of arguments that was not equal to four. Our next sample input consisted of four *Strings*, but the two corresponding to the visitor and acceptor classes were class names that did not appear in the model. For our final test, we sent a list of four *Strings*, but the type for the new attribute to be added was not present in the model. In all cases, the method correctly identified and reported an error in the input.

5.14.3 Execute

To test the *execute* method, we passed it four *Strings* with the values *TaxProcessorVisitor*, *Acceptor*, *InfoClass*, and *Info*. The results of applying the transform to these inputs can be seen in figures 5.22a-b. We note that the *Accept* methods of all classes that contained it have been updated to take an additional argument of type *InfoClass* with the name *info*. In addition, these methods in *EuropeTaxProcessor* and *USTaxProcessor* have been updated to call the *visit* method and pass it additional argument as *null*. We choose to pass *null* and leave it up to the engineer to change the method to pass something more appropriate. Finally, we turn our attention toward the *TaxProcessorVisitor* class and its two *visit* methods. We note that they have also been updated to take the new *InfoClass* argument named *Info*.

```
package visPackage is
   type double is digits 10 range -1.0E-20 .. 1.0E20;
   class EuropeTaxProcessor is Acceptor with
      public class function EuropeTaxProcessor() : EuropeTaxProcessor
         is begin EuropeTaxProcess := new EuropeTaxProcessor; end;
      public function calculateTaxes(total : in double)  : double
         ... continued in Figure 5.33b ...
```

Figure 5.22a: Beginning of AWL generated from executing the visitor transform.

```
... continued from Figure 5.33a ...
     is begin calculateTaxes := 0.15 * total; end;

      public procedure Accept(visitor : in TaxProcessorVisitor, info
: in InfoClass)
         is begin
            visitor.VisitEuropeTaxProcessor(this, null); end;
   end class;

   class USTaxProcessor is Acceptor with

public class function USTaxProcessor()  : USTaxProcessor
         is
         begin
            USTaxProcess := new USTaxProcessor;
         end;
      public function calculateTaxes(total : in double)  : double
         is
         begin
            calculateTaxes := 0.075 * total;
         end;
      public procedure Accept(visitor : in TaxProcessorVisitor, info
: in InfoClass)
         is
         begin
            visitor.VisitUSTaxProcessor(this, null);
         end;
   end class;

   class Acceptor is abstract
      public abstract procedure Accept(visitor : in
TaxProcessorVisitor, info : in InfoClass)
   end class;

   class TaxProcessorVisitor is abstract
      public abstract procedure VisitEuropeTaxProcessor(node : in
EuropeTaxProcessor, info : InfoClass)
      public abstract procedure VisitUSTaxProcessor(node : in
USTaxProcessor, info : InfoClass)
   end class;

   class InfoClass is
      public secretTax : double;
      public class function InfoClass()  : InfoClass
         is
         begin
            secretTax := 0.05;
         end;
   end class;
end package;
```

Figure 5.22b: Continuation of AWL generated from executing the visitor transform.

5.15 Summary

In this chapter, we began by describing our strategy for testing each transform individually. For each transform, we specified an AWL file that would satisfy the *applicable* method for that transform. We then specified the input to the *execute* method and showed that it worked correctly through a visual inspection of the AWL generated by transforming the altered AST. To test the *applicable* method we specified a range of inputs used on a case by case basis. As each of the sections has shown, all transforms are working correctly as specified by the requirements laid out in Chapter 4.

VI. CONCLUSIONS AND FUTURE WORK

To set the stage for our work, we began with a description of AWSOME and many common design patterns. We then took an object oriented approach in designing transformations meant to alter the entire structural model of an existing AST to support the notion of a given design pattern. For each transform we followed a simple software life cycle: requirements specification, design, implementation, and testing. Although this thesis develops a solid framework for transforms of this nature, there is a potential for future work in this area.

6.1 Conclusions

While many of the previously created transforms for AWSOME change the structural model in terms of methods and statements, this thesis takes such changes one step further by adding entire new classes to models. Through this approach, we are able to preserve existing class relationships present in a model while adding new classes and methods to support the notion of a specific design pattern. Previous work regarding AWSOME transforms focused on relatively straightforward steps that took an analysis model to code whereas in this thesis our work changes the entire structure of a system.

6.1.1 Transforms

We began our discussion in Chapter 1 with a general outline of AWSOME and design patterns. We then moved into a deeper discussion of AWSOME, AWL, and specific design patterns (as well as the feasibility of implementing each pattern) in Chapter 2. In Chapter 3 we laid out the requirements that a model transformed to support each pattern would have to meet in order to be considered implementing said pattern. After describing

the layout of a transform in AWSOME, we described the implementation for each design pattern transform in Chapter 4. Finally, we developed our test cases and testing results throughout Chapter 5.

Through our work to achieve the goal of automatically altering a structural model to support a specific design pattern, the following twelve transforms were developed.

1 **Singleton**

(i) XformSing1: Transforms a class into a singleton class.

(ii) XformSing2: Transforms a model to make use of a singleton class.

2 **Abstract Factory**

(i) XformAF1: Transforms a model to contain an abstract factory.

(ii) XformAF2: Transforms a model to make use of an abstract factory.

3 **Factory Method**

(i) XformFactM1: Transforms a model to contain a factory method class.

(ii) XformFactM2: Transforms a model to make use of a factory method class.

4 **Memento**

(i) XformMem1: Transforms a model to contain a memento class and transforms the class to support the memento.

5 **Observer**

(i) XformObs1: Transforms a model to contain the classes and shells of methods necessary to support the observer pattern.

(ii) XformObs2: Transforms a class into an Observer.

6 **Visitor**

(i) XformVis1: Transforms a model by adding a new abstract super class existing classes can inherit from.

(ii) XformVis2: Transforms a model with an existing visitor pattern to allow an additional attribute to be passed to the classes' *accept* methods.

(iii) XformVis3: Transforms a model to support the visitor pattern.

Each group of transforms accomplishes the task of updating an existing AST to support the specified design pattern. Though emphasis is placed on updating the model to support the pattern, each transform also serves to update the AST in such a way that correct code can be generated from the code generator in AWSOME.

6.1.2 Methodology

From our work in this thesis, it is evident that existing software models can be modified in an automatic way to support design patterns. However, in our development process, we have created a strategy that serves to aid in the creation of further such transforms. Our methodology is presented as a series of steps, with each step having various considerations that must be evaluated before continuing. It should be noted that this list of steps precludes choosing a pattern itself; one would expect that a useful pattern to automate has already been selected by the implementer.

1 **Requirements**: The initial step is a relatively simple one, but perhaps the most fundamentally important. We examine a complete implementation of a given design pattern and decide what a final system would look like in terms of classes, attributes, methods, and statements to decide if the pattern could be implemented in our meta-model language. Considerations for this step are pattern requirements such as multiple inheritance, nested classes, static typing,

and other mechanisms that vary among languages.

2 **Determine transform inputs**: In this step, we use input to refer to two ideas. First, we develop the state an existing model must satisfy in order for the pattern to be applied. In a formal sense, this state can be likened to the preconditions for applying the transform. In addition, we specify the literal input to the transform that a user must physically give the transform in order for it to work correctly.

3 **Determine transform outputs**: For this step, we decide what classes, methods, attributes, and statements will be added to a model to accomplish the transform. Formally, the outputs of the transform can be likened to the post-condition of the transform.

4 **Compare inputs/outputs**: In order to be useful, obviously a design pattern transform must create some output. However, usefulness as a measure should also account for the ratio of input to output. In picking a transform to automate, we want to be able to generate a large amount of output with minimal user input. For example, in designing a transform for the adaptor pattern, we note that a user would have to specify which methods the adaptor class calls in each of its methods. Requiring this much input from the user to simply automatically generate method calls in the model is, in our opinion, a poor ratio of input to output.

5 **Design/Implementation/Testing**: Once the first four steps have been completed, the remaining work is the rest of a typical software lifecycle.

6.1.3 Tool Support

We previously discussed the *GumpPanel* in our testing chapter. As it was introduced, it was marketed as a testing harness to show that the transforms designed were working as specified. However, it is not the case that its purpose is solely for testing: *GumpPanel* is complete in the sense that a user could utilize it to apply any of our transforms to his or her own AST. In order to do so, a user must first use the *ToolLoader* (accessible from the main menu of AWSOME) to parse his AWL file into an AST. Next, the user can select *Tool2009* (also on the main menu) to load the interface to all transforms. With this screen open, the user need only select the tab labeled "Gump" and then any of our transforms can be applied by simply clicking the appropriate button.

6.2 Future Work

While this thesis has made advances in the way of creating transforms that update structural models to support design patterns, there is a potential for future work to be conducted in this area. The range of possibilities for future work include enhancing the transforms given by this thesis, adding new design pattern transforms, and changing some aspects of AWSOME itself.

While the transforms presented in this thesis are complete in the sense of their *applicable* and *execute* methods, we, like previous transform developers for AWSOME, have omitted implementation of two methods also present in the *transform* class: *unDo* and *replay*. In the case of *unDo*, a detailed history syntax would need to be developed in order to indentify precisely what changes were made to the model. Since some of the transforms in this thesis directly change existing model structure and individual statements, we would require that this history record the initial state of the target of the

change as well as the new state. While this would require making a complicated system for storing histories, once a scheme is developed it could play a large role in the *replay* operation. In essence, while *unDo* would focus on reversing the changes in the history, *replay* could simple reapply all the changes listed.

While the previous section focuses on creating a means of storing and loading a history, we also note that some of the transforms included in this thesis could also be expanded on. In some cases, our transforms created classes and relevant methods to support a design pattern, but the bodies of some methods were left empty. The rationale for leaving them empty is that we do not have a priori knowledge what the methods should do. For example, in the observer transform, we create an *observer* class with an abstract *update* method that must be implemented by concrete subclasses. Furthermore, we provide a transform to create a new *observer* subclass in the model, but we leave its *update* method body blank. Future work could involve allowing a user to indicate some behavior for this function such as a method call. On the same token, our visitor transform sets up the required class hierarchy and methods that would allow new visitor classes to be created and take advantage of the pattern. However, we do not create any concrete visitor classes because we do not know what they should do in their *visit* methods. Despite this, future work could create a new *visitor* subclass class such as one that can be used to visit every node in an inheritance tree. Another transform that holds potential for future work is our visitor transform that creates a new abstract super class. Work in this area could involve allowing new methods and attributes to be specified, forcing subclasses to automatically include the methods, set to return dummy values (though this still leaves the body of the method essentially empty).

In completing this thesis, we focused primarily on many well known design patterns that initially appeared in the book Design Patterns: Elements of Reusable Object-Oriented Software [5]. While Chapter 2 details our rationale as to why particular patterns were not developed into transforms for this thesis, we also note that time constraints prevented us from exploring other patterns that are not in the original book. For example, we include a transform for the singleton pattern, but we did not also include one for the multiton (allow a set number of instances of a class rather than only one) [15]. Furthermore, our conclusions on which patterns were feasible or beneficial to implement should not necessarily be taken as an end all discussion. On the contrary, one may find techniques to minimize the amount of input required to automate a transform and thus make its implementation worthwhile.

While our emphasis has been on expanding on the work of transforms done in this thesis, we do not overlook the fact that AWSOME has some components that could benefit from improvement. A primary concern for future work is that of the parser, as at the time of this writing it does not support the notion of static attributes. This limitation is detrimental to some of the work in this thesis: any transform in which we create a static attribute will lose the notion of being static if we re-write the model as an AWL file. Clearly we would like to be able to save this information if we are to continue our support of being able to parse an AWL file into an AST so that the system can operate on it. Furthermore, while we updated the Java code generator to handle correctly translating instantiation of objects using the *new* operator and a default constructor, it still requires updating to allow instantiation using a constructor that takes arguments.

6.3 Summary

Through our work in this thesis, we have developed many useful transforms that show the feasibility of creating transforms that apply design patterns to existing software models. In implementing these transforms, we have also made changes to other parts of AWSOME such as the code generator as well as developed a strategy for creating new design pattern transforms. With the solid ground work this thesis has laid, we are confident that future work will lead to a comprehensive, robust library of design pattern transforms for AWSOME.

APPENDIX A

Java Code

The following section contains the Java code generated by applying the Java code generator to the AWL presented in chapter 5's testing section. The AWL used is the direct result of applying a transform's *execute* method to the input AWL file. For each piece of presented code, the transform it is associated with is noted. This appendix is included as an easier way to visually detect that a transform has been correctly applied rather than decipher that fact from the AWL.

XformSing1 Transform

As noted, the *singleton* attribute is typed as *static*, as it should be to support the singleton pattern. In addition, due to a previous design decision in AWSOME, private attributes become protected with turned into Java code.

```
package simpTax;
/**
* File    CanadaTaxProcesor.java
*/
public class CanadaTaxProcessor {
  protected    double rate;
  protected static    CanadaTaxProcessor singleton;

  protected  CanadaTaxProcessor() {
    rate=0.05;
  }

  public double calculateTaxes(double total) {
    return rate*total;
  }

  public static CanadaTaxProcessor getInstance() {
    if (singleton == null) {
      singleton= new CanadaTaxProcessor();
    }
    return singleton;
  }

}
```

XformSing2 Transform

In order to enhance clarity, we show only the *CanadaTaxUser* class, which makes use of the previously shown *CanadaTaxProcessor* class.

```
package simpTax;
/**
* File    CanadaTaxUser.java
*/
public class CanadaTaxUser {
  protected    CanadaTaxProcessor theTaxer;

  public  CanadaTaxUser() {
  }
  public void testTaxer() {
    double rate;
    theTaxer=CanadaTaxProcessor.getInstance();
    theTaxer.calculateTaxes(20.0);
    rate=1;
    if (5 > 6) {
      theTaxer=CanadaTaxProcessor.getInstance();
    } else {
      if (6 < 5) {
        theTaxer=CanadaTaxProcessor.getInstance();
      }
    }
    while (rate < 5) {
      if (rate == 3) {
        theTaxer=CanadaTaxProcessor.getInstance();
      } else {
        rate=rate+1;
      }
    }
  }
  public double testTax() {
    CanadaTaxProcessor theTaxer;
    theTaxer=CanadaTaxProcessor.getInstance();
    return 0.05;
  }
}
```

XformAF1 Transform

In order to enhance clarity, we show only the *AbstractFactoryW* class and one of the concrete subclasses, *ConcreteFactoryS*, noting that the other two are similar. The classes that existed before the transform are identical and also not shown..

```
package taxPackage;
/**
* File    AbstractFactoryW.java
*/
public abstract class AbstractFactoryW {

  public abstract TaxProcessor createTaxProcessor();
  public abstract SalaryCalc createSalaryCalc();
  public static AbstractFactoryW getFactory(Integer factoryChoice) {

    if (factoryChoice == 1)
      {  return new ConcreteFactoryS(); }
    if (factoryChoice == 2)
      {  return new ConcreteFactoryLGN(); }
    if (factoryChoice == 3)
      {  return new ConcreteFactoryHMOXA();  }
  }
}

package taxPackage;
/**
* File    ConcreteFactoryS.java
*/
public class ConcreteFactoryS extends AbstractFactoryW {

  public TaxProcessor createTaxProcessor() {
    return new EuropeTaxProcessor();
  }

  public SalaryCalc createSalaryCalc() {
    return new EuropeSalaryCalc();
  }
}
```

XformAF2 Transform

For clarity, we show only the updated *User* class as the rest of the Java code is identical to the *XformAF1 Transform* and the *User2* class is similar.

```
package taxPackage;
/**
* File    User.java
*/
public class User {
  protected   TaxProcessor taxer;
  protected   SalaryCalc calcer;
  protected   AbstractFactoryW theAbstractFactoryW;

  public  User() {
  }

  public void tester() {
    theAbstractFactoryW=AbstractFactoryW.getFactory(3);
    taxer=theAbstractFactoryW.createTaxProcessor();
    calcer=theAbstractFactoryW.createSalaryCalc();

    if (5 > 6) {
      calcer=theAbstractFactoryW.createSalaryCalc();
    } else {
      if (6 < 5) {
        taxer=theAbstractFactoryW.createTaxProcessor();
      }
    }
    taxer.calculateTaxes(25);
    calcer.calcSalary(40);
  }
}
```

XformFactM1 Transform

In this case, we show only the *TaxProcessorFactory* class as it is the only change present in the model. As we can see, this has the look of a standard Java file that would compile.

```
package taxPackage;
/**
* File    TaxProcessorFactory.java
*/
public class TaxProcessorFactory {
  protected    boolean EuropeTaxProcessorFlag;
  protected    boolean CanadaTaxProcessorFlag;
  protected    boolean USTaxProcessorFlag;

  public  TaxProcessorFactory() {
  }

  public TaxProcessor getInstance() {
    TaxProcessor tmp;
    if (EuropeTaxProcessorFlag == true)
      {  tmp=new EuropeTaxProcessor(); }
    if (CanadaTaxProcessorFlag == true)
      {  tmp=new CanadaTaxProcessor(); }
    if (USTaxProcessorFlag == true)
      {  tmp=new USTaxProcessor(); }
    return tmp;
  }

  public void useEuropeTaxProcessor() {
    EuropeTaxProcessorFlag=true;
    EuropeTaxProcessorFlag=false;
    EuropeTaxProcessorFlag=false;
  }

  public void useCanadaTaxProcessor() {
    CanadaTaxProcessorFlag=false;
    CanadaTaxProcessorFlag=true;
    CanadaTaxProcessorFlag=false;
  }

  public void useUSTaxProcessor() {
    USTaxProcessorFlag=false;
    USTaxProcessorFlag=false;
```

```
    USTaxProcessorFlag=true;
  }
}
```

XformFactM2 Transform

To highlight the relevant changes, we show only the *CanadaTaxUser* class. As this code file shows syntactically correct Java (assuming the referenced classes are in the same package, though not shown), we see that working code can be generated from the AWL created by the transform.

```
package taxPackage;
/**
* File    CanadaTaxUser.java
*/

public class CanadaTaxUser {
  protected    TaxProcessor theTaxer;
  protected    TaxProcessorFactory theTaxProcessorFactory;

  public  CanadaTaxUser() {
    theTaxProcessorFactory=new TaxProcessorFactory();
  }

  public void testTaxer() {
    double rate;
    theTaxProcessorFactory.useCanadaTaxProcessor();
    theTaxer=theTaxProcessorFactory.getInstance() ;
    theTaxer.calculateTaxes(20.0);
    rate=1;
    if (5 > 6) { theTaxer=theTaxProcessorFactory.getInstance() ; }
    else {
      if (6 < 5) { theTaxer=theTaxProcessorFactory.getInstance() ;}
    }
    while (rate < 5) {
      if (rate == 3) {
        theTaxer=theTaxProcessorFactory.getInstance() ;
      } else { rate=rate+1; }
    }
  }

  public double testTax() {
    EuropeTaxProcessor theTaxer2;
    theTaxProcessorFactory.useEuropeTaxProcessor();
    theTaxer2=theTaxProcessorFactory.getInstance() ;
    return 0.05;
  }
```

```
}
```

XformMem1 Transform

To enhance clarity, we display only the modified *CanadaTaxProcessor* and *CanadaTaxProcessorMemento* classes. While we note that the notion of the array did not clearly translate to the Java code, this issue is not related to our transform. In fact, we see that all relevant components of our transform became syntactically correct Java code.

```
package simpTax;
/**
* File    CanadaTaxProcessorMemento.java
*/
public class CanadaTaxProcessorMemento {
  protected    double rate;
  protected    otherClass other;
  protected    int num;
  protected    TArray arr;
  protected    TPtr arrP;

  public CanadaTaxProcessorMemento(double rate1, otherClass other1, int num1, TArray
arr1, TPtr arrP1) {
    rate=rate1;
    other=other1;
    num=num1;
    arr=arr1;
    arrP=arrP1;
  }

  public double getrate() { return rate;  }

  public otherClass getother() { return other; }

  public int getnum() { return num; }

  public TArray getarr() { return arr; }

  public TPtr getarrP() { return arrP; }

  public void setrate(double therate) {
    rate=therate;
  }

  public void setother(otherClass theother) {
    other=theother;
```

```
  }

  public void setnum(int thenum) {
    num=thenum;
  }

  public void setarr(TArray thearr) {
    arr=thearr;
  }

  public void setarrP(TPtr thearrP) {
    arrP=thearrP;
  }
}

package simpTax;
/**
* File    CanadaTaxProcessor.java
*/

public class CanadaTaxProcessor {
  protected    double rate;
  protected    otherClass other;
  protected    int num;
  protected    TArray arr;
  protected    TArray [] arrP;

  public  CanadaTaxProcessor() {
    rate=0.05;
  }

  public double calculateTaxes(double total) {
    double sample;
    return rate*total;
  }

  public TArray copyarr() {
    return arr;
  }

  public TArray [] copyarrP() {
    return arrP;
  }

  public                void                setMemento(CanadaTaxProcessorMemento
theCanadaTaxProcessorMemento) {
```

```
    rate=theCanadaTaxProcessorMemento.getrate() ;
    other=theCanadaTaxProcessorMemento.getother() ;
    num=theCanadaTaxProcessorMemento.getnum() ;
    arr=theCanadaTaxProcessorMemento.getarr() ;
    arrP=theCanadaTaxProcessorMemento.getarrP() ;
  }

  public CanadaTaxProcessorMemento getMemento() {
    CanadaTaxProcessorMemento localMem;
    double localrate;
    otherClass localother;
    int localnum;
    TArray localarr;
    TPtr localarrP;

    localMem=new CanadaTaxProcessorMemento();
    localrate=rate;
    localother=other.copy() ;
    localnum=num;
    localarr=copyarr() ;
    localarrP=copyarrP() ;
    localmem.setrate(localrate);
    localmem.setother(localother);
    localmem.setnum(localnum);
    localmem.setarr(localarr);
    localmem.setarrP(localarrP);
    return localMemento;
  }
}
```

XformObs1 Transform

In this case, we created two new classes so both are displayed in the diagram, while the original class is omitted. While we acknowledge that Java already has its own set of classes appropriate for applying the observer pattern, we chose not to use them to provide a language independent solution. While we do not create any subclasses to implement the abstract classes' methods, the code is syntactically correct.

```
package simpTax;
/**
* File    Observer.java
*/
public abstract class Observer {

  public abstract void update();
}

package simpTax;
/**
* File    Observable.java
*/
public abstract class Observable {

  public void attach(Observer o) {}

  public void detach(Observer o) {}

  public void notify() { }
}
```

XformObs2 Transform

Since the only class to change was *CanadaTaxProcessor*, it is the only class file shown. As expected, syntactically correct Java is produced (assuming the *Observer* and *Observable* class are in the same package). The *update* method is blank, left for an engineer to decide how a *CanadaTaxProcessor* should react to an update, but having it blank does not violate Java syntax since its return type is *void*.

```
package simpTax;
/**
* File    CanadaTaxProcessor.java
*/

public class CanadaTaxProcessor extends Observer {
  protected    double rate;
  protected    int num;

  public  CanadaTaxProcessor() {
    rate=0.05;
  }

  public double calculateTaxes(double total) {
    double sample;
    return rate*total;
  }

  public void update() {
  }
}
```

XformVis1 Transform

Rather than transform all of the files into separate Java class files to show that correct code can be generated, we only transform the *Acceptor* class. The other classes transform as expected, simply become class files that *extend* the *Acceptor* class. As expected, there is very little to the class, but it does translate to syntactically correct Java code.

```
package visPackage;
/**
* File    Acceptor.java
*/
public abstract class Acceptor {
}
```

XformVis3 Transform

We opt to only show a few classes, noting the others are similar. In this case, we show only the transformed Java code for the *Acceptor*, *TaxProcessorVisitor*, and *USTaxProcessor* classes. We note that all the classes and methods correctly translated to syntactically correct Java code, as expected.

```
package visPackage;
/**
* File    USTaxProcessor.java
*/
public class USTaxProcessor extends Acceptor {

  public  USTaxProcessor() {
    USTaxProcess=new USTaxProcessor();
  }

  public double calculateTaxes(Double total) {
    return 0.075*total;
  }

  public void Accept(TaxProcessorVisitor visitor) {
    visitor.VisitUSTaxProcessor(this);
  }
}

package visPackage;
/**
* File    Acceptor.java
*/
public abstract class Acceptor {

  public abstract void Accept(TaxProcessorVisitor visitor);
}

package visPackage;
/**
* File    TaxProcessorVisitor.java
*/
public abstract class TaxProcessorVisitor {

  public abstract void VisitEuropeTaxProcessor(EuropeTaxProcessor node);
```

```
  public abstract void VisitUSTaxProcessor(USTaxProcessor node);
}
```

XformVis2 Transform

In order to enhance clarity, we show only the *TaxProcessorVisitor*, *Acceptor*, and *USSalaryProcessor* classes, and note that the other classes transformed similarly. As expected, the classes produced are syntactically correct and can be expanded upon to take advantage of the visitor pattern and the newly added information attribute.

```
package visPackage;
/**
* File    Acceptor.java
*/
public abstract class Acceptor {

  public abstract void Accept(TaxProcessorVisitor visitor, InfoClass info);
}

package visPackage;
/**
* File    TaxProcessorVisitor.java
*/
public abstract class TaxProcessorVisitor {

  public abstract void VisitEuropeTaxProcessor(EuropeTaxProcessor node, InfoClass
info);

  public abstract void VisitUSTaxProcessor(USTaxProcessor node, InfoClass info);
}

package visPackage;
/**
* File    USTaxProcessor.java
*/
public class USTaxProcessor extends Acceptor {


  public  USTaxProcessor() {
    USTaxProcess=new USTaxProcessor();
  }

  public double calculateTaxes(double total) {
    return 0.075*total;
  }
```

```
  public void Accept(TaxProcessorVisitor visitor, InfoClass info) {
    visitor.VisitUSTaxProcessor(this, null);
  }
}
```

APPENDIX B

Java Code

The following section contains Java code that was written to accomplish each of our transforms. Rather than include all of the code in each transform's Java file, we include only snippets from the *execute* and *applicable* methods. In some cases, these methods use other defined functions and as such we will include these methods as well when appropriate. The final piece of code included is our entire *GumpPanel.java* testing harness file.

XformSing1 Transform

```
/**
 * Source file: XformSing1.java
 * Purpose: apply the basic singleton design pattern
 * <pre>
 * History:
 *    Original: 06-23-09 Gump created including methods:
 *              -XformSing1() default constructor
 *              -XformSing1(WsPackage) set up the ast
 *              -applicable(Object, OBject) check if can be applied
 *              -explain(Object) explain what will happen if can be applied
 *              -getDesc() description of transform
 *              -getName() name of xform
 *              -replay(WsPackage) not implemented
 *              -show() not implemented
 *              -toString() nothing really
 *              -undo(WsPackage, Object) not implemented
 *    Modified:
 * </pre>
 */

/**
    * Checks if singleton can be applied.  To be applied, the tgt must be
    * a valid ast and the params must be a class to apply singleton pattern
    * to.  In order to apply the pattern, the class must not contain a method
    * named getInstance() and must not contain an attribute named singleton.
    * In addition, a default no-arg constructor is expected.
    * @param tgt the ast to be modified
    * @param params the list of lists
    * @return whether or not the pattern can be applied
    */
   public static boolean applicable(Object tgt, Object params)
   {
      if ( tgt == null )
         return false;
      if ( !(params instanceof WsClass))
         return false;
      WsClass cls = (WsClass)params;
      Vector<WsMethod> meths = cls.getWsClassOperations();
      //checking for method
      boolean defaultConstructor = false;
      for ( int i = 0; i < meths.size(); i++)
      {
         //named getInstance
         if ( meths.get(i).getName().equals("getInstance") )
```

```
        {
            //no arguments
            if ( meths.get(i).getFormals() == null || meths.get(i).getFormals().size() == 0)
            {
                System.out.println("getInstance() already exists!");
                return false;
            }
        }
        else if ( meths.get(i).getName().equals(cls.getName()) )
        {
            if ( meths.get(i).getFormals() == null || meths.get(i).getFormals().size() == 0)
            {
                defaultConstructor = true;
            }
        }
    }
    if ( !defaultConstructor )
    {
        System.out.println("No default constructor found.");
        return false;
    }

    //get all the class attributes
    Vector<WsAttribute> attrs = cls.getWsClassDataComponents();
    for ( int i = 0; i < attrs.size(); i++)
    {
        //check name of attribute for singleton
        if ( attrs.get(i).getName().equals("singleton") )
        {
            System.out.println("singleton attribute already exists!");
            return false;
        }
    }
    return true;
}

/**
 * Applies the singleton to the existing model passed into constructor.
 * Does so by creating a static attribute as the same type of class in the class
 * named singleton.  Also creates a function to control instantiating and
 * returning the singleton variable named getInstance().
 * @param params the class to be transformed into a singleton
 * @return success or failure
 */
@Override
public boolean execute(Object params)
```

```
{
   if ( !applicable(target, params) )
      return false;

   //turn into a class because it should be!
   WsClass cls = (WsClass)params;

   WsFunction m = new WsFunction();
   WsMethod getInstance = new WsMethod(m);
   getInstance.setWsMethodSubprogram(m);
   m.setName("getInstance");
   String identName = cls.getName();
   WsIdentifierRef returnIdent = new WsIdentifierRef(identName);
   //returnIdent.setWsIdentRefTo(cls.getWsDeclName());
   //System.out.println(cls.getWsDeclName());
   m.setWsFunctReturnType(returnIdent);
   getInstance.setWsAbstract(false);
   getInstance.setWsClassMethod(true);
   getInstance.setWsPrivate(false);
   getInstance.setFunctReturnType(identName);
   cls.addWsClassOperation(getInstance);

   //create the attribute itself
   WsAttribute singleton = new WsAttribute();
   singleton.setAttributeName("singleton");
   singleton.setWsStatic(true);
   singleton.setAttributeTypeName(identName);
   singleton.setWsPrivate(true);
   cls.addWsClassDataComponent(singleton);

   //create the logic inside of getInstance()
   WsIdentifierRef tmp;
   WsSelection ifElse = new WsSelection();
   WsExpression equals = new WsEqual();
   //left side variable name, singleton
   tmp = new WsIdentifierRef(singleton.getName());
   tmp.setWsIdentRefTo(new WsIdentifier(singleton.getName()));
   ((WsEqual)equals).setWsBinExpOp1(tmp);
   //right side, value to if null
   ((WsEqual)equals).setWsBinExpOp2(new WsLiteralNull());
   ifElse.setWsSelCondition(equals);
   WsAssignment newS = new WsAssignment();
   tmp = new WsIdentifierRef(singleton.getName());
   tmp.setWsIdentRefTo(new WsIdentifier(singleton.getName()));
   newS.setWsAssignLHS(tmp);
   //WsExpression rhs = new WsFunctionCall(cls.getName());
```

```
    WsAllocator rhs = new WsAllocator();
    tmp = new WsIdentifierRef(cls.getName());
    tmp.setWsIdentRefTo(cls.getWsName());
    rhs.setWsAllocReturnType(tmp);
    newS.setWsAssignRHS(rhs);
    //add if part, else is unused
    ifElse.addWsSelThenPart(newS);
    m.addWsSubprogBody(ifElse);
    //do return statement part
    WsAssignment bod = new WsAssignment();
    tmp = new WsIdentifierRef(m.getName());
    tmp.setWsIdentRefTo(m.getWsName());
    bod.setWsAssignLHS(tmp);
    tmp = new WsIdentifierRef(singleton.getName());
    tmp.setWsIdentRefTo(new WsIdentifier(singleton.getName()));
    WsExpression r = tmp;
    bod.setWsAssignRHS(r);
    m.addWsSubprogBody(bod);

    //make the constructor private
    Vector<WsMethod> meths = cls.getWsClassOperations();
    for ( int i = 0; i < meths.size(); i++)
    {
       if ( meths.get(i).getName().equals(cls.getName()) )
       {
          if ( meths.get(i).getFormals() == null || meths.get(i).getFormals().size() == 0)
          {
             meths.get(i).setWsPrivate(true);
          }
       }
    }

    return true;
}
```

XformSing2 Transform

```
/**
 * Source file: XformSing2.java
 * Purpose: update the model to use the applied singleton model
 * <pre>
 * History:
 *    Original: 08-08-09 Gump created including methods:
 *              -XformSing2() default constructor
 *              -XformSing2(WsPackage) set up the ast
 *              -applicable(Object, OBject) check if can be applied
 *              -explain(Object) explain what will happen if can be applied
 *              -getDesc() description of transform
 *              -getName() name of xform
 *              -replay(WsPackage) not implemented
 *              -show() not implemented
 *              -toString() nothing really
 *              -undo(WsPackage, Object) not implemented
 *    Modified:
 * </pre>
 */

/**
    * Checks if singleton2 can be applied.  To be applied, the class specifed as
    * an argument must have already been transformmed into a singleton.  That is
    * to say, it has an attribute of itself declared as static, named singleton
    * and has a function named getInstance that returns that type.
    * @param tgt the ast to be modified
    * @param params the class that is a singleton
    * @return whether or not the pattern can be applied
    */
   public static boolean applicable(Object tgt, Object params)
   {
      if ( tgt == null )
         return false;
      if ( !(params instanceof WsClass))
         return false;
      WsClass cls = (WsClass)params;
      Vector<WsMethod> meths = cls.getWsClassOperations();
      //checking for method
      boolean defaultConstructor = false;
      boolean getInstance = false;
      boolean singletonVar = false;
      for ( int i = 0; i < meths.size(); i++)
      {
         //named getInstance
```

```
        if ( meths.get(i).getName().equals("getInstance") &&
meths.get(i).getWsClassMethod())
        {
           //no arguments
           if ( meths.get(i).getFormals() == null || meths.get(i).getFormals().size() == 0)
           {
              getInstance = true;
           }
        }
        if ( meths.get(i).getName().equals(cls.getName()) )
        {
           if ( meths.get(i).getFormals() == null || meths.get(i).getFormals().size() == 0)
           {
              defaultConstructor = true;
           }
        }
     }
     Vector<WsAttribute> attrs = cls.getWsClassDataComponents();
     for ( int i = 0; i < attrs.size(); i++ )
     {
        if ( attrs.get(i).getName().equals("singleton") &&
             attrs.get(i).getWsStatic() &&
             attrs.get(i).getTypeName().equals(cls.getName()))
           singletonVar = true;
     }
     if ( !defaultConstructor || !getInstance || !singletonVar)
     {
        System.out.println("No default constructor/getInstance/singleton variable
found.");
        return false;
     }
     return true;
  }

  /**
   * Applies the singleton2 to the existing model passed into constructor.
   * Does so by creating updating all instansiating of the class with calls
   * to the factory's static method that returns an instance.
   * @param params the class to be transformed into a singleton
   * @return success or failure
   */
  @Override
  public boolean execute(Object params)
  {
     if ( !applicable(target, params) )
        return false;
```

```
        //turn into a class because it should be!
        WsClass cls = (WsClass)params;

        //get all classes in the model
        Vector<WsDeclaration> decls = target.getWsDecls();
        Vector<WsClass> theClasses = new Vector<WsClass>();
        for ( int i = 0; i < decls.size(); i++ )
        {
           if ( decls.get(i) instanceof WsClass )
              theClasses.add((WsClass)decls.get(i));
        }
        theClasses.remove(cls);
        //make the function call we will constantly reuse
        WsFunctionCall getFact = new WsFunctionCall(cls.getName() + ".getInstance");

        //update each of the classes
        for ( int i = 0; i < theClasses.size(); i++ )
        {
           Vector<WsMethod> meths = theClasses.get(i).getWsClassOperations();

           //check each method
           for ( int j = 0; j < meths.size(); j++ )
           {
              Vector<WsStatement>                                    stmts                                    =
meths.get(j).getWsMethodSubprogram().getWsSubprogBody();
              allStmts = new Vector<WsStatement>();
              //check all the statements
              for ( int k = 0; k < stmts.size(); k++ )
              {
                 //only these two have to be searched recursively because
                 //they could have the selects/iters in their body
                 if ( stmts.get(k) instanceof WsSelection)
                 {
                   recursiveFindSelStmts((WsSelection)stmts.get(k));
                 }
                 else if ( stmts.get(k) instanceof WsIteration)
                 {
                    recursiveFindIterStmts((WsIteration)stmts.get(k));
                 }
                 else
                    allStmts.add(stmts.get(k));
              }
              for ( int k = 0; k < allStmts.size(); k++)
              {
                 if ( allStmts.get(k) instanceof WsAssignment)
```

```
                {
                  WsAssignment s = (WsAssignment)allStmts.get(k);
                  //is this something = new somethingElse?
                  if ( s.getWsAssignRHS() instanceof WsAllocator)
                  {
                     WsAllocator alloc = (WsAllocator)s.getWsAssignRHS();
                     //is this an instance of our singleton class?
                     if ( alloc.getWsAllocReturnType().getName().equals(cls.getName()))
                        //replace it with our function call instead!
                        s.setWsAssignRHS(getFact);
                  }
                }
             }
          }
       }

       return true;
    }

/**
     * Recursively explores if/elses to fine any nested statements
     * @param wsSelection initial if/else to search
     */
    private void recursiveFindSelStmts(WsSelection wsSelection)
    {
       Vector<WsStatement> stmtsE = wsSelection.getWsSelElsePart();
       Vector<WsStatement> stmtsT = wsSelection.getWsSelThenPart();

       for ( int i = 0; i < stmtsE.size(); i++)
       {
          if ( stmtsE.get(i) instanceof WsSelection)
             recursiveFindSelStmts((WsSelection)stmtsE.get(i));
          else if ( stmtsE.get(i) instanceof WsIteration)
             recursiveFindIterStmts((WsIteration)stmtsE.get(i));
          else
             allStmts.add(stmtsE.get(i));
       }
       for ( int i = 0; i < stmtsT.size(); i++)
       {
          if ( stmtsT.get(i) instanceof WsSelection)
             recursiveFindSelStmts((WsSelection)stmtsT.get(i));
          else if ( stmtsT.get(i) instanceof WsIteration)
             recursiveFindIterStmts((WsIteration)stmtsT.get(i));
          else
             allStmts.add(stmtsT.get(i));
       }
```

```
}

/**
 * Recursively searching while loops to find nested statements in them
 * @param wsIteration initial while loop to search
 */
private void recursiveFindIterStmts(WsIteration wsIteration)
{
   Vector<WsStatement> stmtsE = wsIteration.getWsIterBody();

   for ( int i = 0; i < stmtsE.size(); i++)
   {
      if ( stmtsE.get(i) instanceof WsSelection)
         recursiveFindSelStmts((WsSelection)stmtsE.get(i));
      else if ( stmtsE.get(i) instanceof WsIteration)
         recursiveFindIterStmts((WsIteration)stmtsE.get(i));
      else
         allStmts.add(stmtsE.get(i));
   }
}
```

XformAF1 Transform

```
/**
 * Source file: XformAF1.java
 * Purpose: apply the basic abstract factory design pattern
 * <pre>
 * History:
 *    Original: 05-01-09 Gump created including methods:
 *              -XformAF1() default constructor
 *              -XformAF1(WsPackage) set up the ast
 *              -applicable(Object, OBject) check if can be applied
 *              -explain(Object) explain what will happen if can be applied
 *              -getDesc() description of transform
 *              -getName() name of xform
 *              -replay(WsPackage) not implemented
 *              -show() not implemented
 *              -toString() nothing really
 *              -undo(WsPackage, Object) not implemented
 *    Modified:
 * </pre>
 */

/**
    * Checks if abstract factory can be applied.  To be applied, the params
    * must be a list of lists, where the first item is an abstract super class
    * of some product and subsequent elements are concrete "themes" of that super
    * class object.
    * @param tgt the ast to be modified
    * @param params the list of lists
    * @return whether or not the pattern can be applied
    */
   public static boolean applicable(Object tgt, Object params)
   {
      //must be a vector of vectors of class
      if ( !(params instanceof Vector))
         return false;
      Vector<Vector<WsClass>> classes;

      //should be vector of vectors classes
      try
      {
         classes = (Vector<Vector<WsClass>>)params;
      }
      catch (Exception e)
      {
         return false;
```

```
    }

    //doesn't serve to transform empty set of classes
    if ( classes.size() == 0 || classes.get(0).size() == 0)
       return false;

    //must be the same number of products for each type
    for ( int i = 0; i < classes.size()-1; i++)
       if ( classes.get(i).size() != classes.get(i+1).size())
          return false;

    //must be abstract super class, products have it as parents
    for ( int i = 0; i < classes.size(); i++)
    {
       WsClass parent = classes.get(i).get(0);
       if ( !parent.getWsClassAbstract())
          return false;
       for ( int j = 1; j < classes.get(i).size(); j++)
       {
          WsClass child = classes.get(i).get(j);
          if ( child.getSuperclassName() == null ||
!(child.getSuperclassName().equals(parent.getName())))
             return false;
       }
    }

    //classes must have default constructor
    for ( int i = 0; i < classes.size(); i++ )
    {
       for ( int j = 1; j < classes.get(i).size(); j++ )
       {
          Vector<WsMethod> methods = classes.get(i).get(j).getWsClassOperations();
          boolean defaultCons = false;
          for ( int k = 0; k < methods.size(); k++)
          {
             WsMethod meth = methods.get(k);
             //must be a constructor, so same name as class
             if ( meth.getName().equals(classes.get(i).get(j).getName()))
             {
                //must have no args
                if ( meth.getFormals() != null || meth.getFormals().size() != 0)
                {
                   defaultCons = true;
                }
             }
             if ( !defaultCons )
```

```
                return false;
            }
        }
    }
    return true;
}

/**
 * Applies the abstract factory to the existing model passed into constructor.
 * Does so by creating an abstract factory super class and concrete subclasses
 * that can return types of related objects as specified by the params.
 * @param params a list of lists, where the sub list have the first item as
 * an abstract product super class and the subsequent elements are subclasses
 * of the super class representing a "theme" of that object.
 * @return success or failure
 */
@Override
public boolean execute(Object params)
{
    //don't bother trying to apply if not applicable
    if ( !applicable(target, params))
        return false;

    //create the abstract super class
    WsClass abstractFactory = new WsClass();
    abstractFactory.setName(XformUtils.makeUniqueClassName(target,
"AbstractFactory"));
    abstractFactory.setWsClassAbstract(true);

    //now we have to create the methods for our class
    //type cast the object to what it really is
    Vector<Vector<WsClass>> groups = (Vector<Vector<WsClass>>)params;
    Vector<WsMethod> theAFOps = new Vector<WsMethod>();
    //createProduct() abstract function construction
    for ( int i = 0; i < groups.size(); i++)
    {
        //first one in each list is the super class name
        WsClass product = (WsClass)(groups.get(i).get(0));
        WsFunction m = new WsFunction();
        WsMethod getProductFunc = new WsMethod(m);
        getProductFunc.setWsMethodSubprogram(m);
        m.setName("create" + product.getName());
        String identName = product.getName();
        WsIdentifierRef returnIdent = new WsIdentifierRef(identName);
        returnIdent.setWsIdentRefTo(product.getWsDeclName());
        m.setWsFunctReturnType(returnIdent);
```

```
   getProductFunc.setWsAbstract(true);
   getProductFunc.setWsClassMethod(false);
   getProductFunc.setWsPrivate(false);
   theAFOps.add(getProductFunc);
}


//create the subclasses of AF
Vector<WsClass> AFSubclasses = new Vector<WsClass>();
for ( int i = 1; i < groups.get(0).size(); i++)
{
   //make the subclass
   WsClass sub = new WsClass();
   sub.setName(XformUtils.makeUniqueClassName(target, "ConcreteFactory"));
   sub.setParent(abstractFactory);
   sub.setSuperclassName(abstractFactory.getName());
   Vector<WsMethod> subMethods = new Vector<WsMethod>();
   for ( int j = 0; j < groups.size(); j++)
   {
      //get the super class
      WsClass product = (WsClass)(groups.get(j).get(0));


      WsFunction m = new WsFunction();
      WsMethod getProductFunc = new WsMethod(m);
      m.setName("create" + product.getName());
      String identName = product.getName();
      WsIdentifierRef returnIdent = new WsIdentifierRef(identName);
      returnIdent.setWsIdentRefTo(product.getWsDeclName());
      m.setWsFunctReturnType(returnIdent);
      getProductFunc.setWsPrivate(false);
      getProductFunc.setWsAbstract(false);
      getProductFunc.setWsClassMethod(false);


      //get an actual subclass that product is of concrete type
      product = (WsClass)(groups.get(j).get(i));
      //write body of method equivalent of return new object
      //i.e. funcName = new ReturnType
      WsAssignment bod = new WsAssignment();
      WsIdentifierRef t = new WsIdentifierRef(getProductFunc.getName());
      t.setWsIdentRefTo(new WsIdentifier(getProductFunc.getName()));
      bod.setWsAssignLHS(t);
      WsExpression rhs = new WsAllocator();
      t = new WsIdentifierRef(product.getName());
      t.setWsIdentRefTo(new WsIdentifier(product.getName()));
```

```
        ((WsAllocator)rhs).setWsAllocReturnType(t);
        bod.setWsAssignRHS(rhs);
        m.addWsSubprogBody(bod);
        subMethods.add(getProductFunc);
    }
    sub.setWsClassOperations(subMethods);
    AFSubclasses.add(sub);
}

//now that we know all of the subclasses, we can make the getFactory method
//for the factory
WsFunction getFactory = new WsFunction();
WsMethod m = new WsMethod(getFactory);
getFactory.setName("getFactory");
String identName = abstractFactory.getName();
WsIdentifierRef returnIdent = new WsIdentifierRef(identName);
returnIdent.setWsIdentRefTo(abstractFactory.getWsDeclName());
getFactory.setWsFunctReturnType(returnIdent);
m.setWsAbstract(false);
m.setWsPrivate(false);
m.setWsClassMethod(true);
//argument list will be an int specifying desired factory
WsIntegerType integer;

//is an integer type already defined in this context?
if (XformUtils.getDeclByName(target, "Integer") instanceof WsIntegerType)
    integer = (WsIntegerType)XformUtils.getDeclByName(target, "Integer");
else
{
    integer = new WsIntegerType();
    integer.setWsDeclName(new WsIdentifier("Integer"));
}


WsParameter arg = new WsParameter("factoryChoice", integer);
arg.setWsParameterIn(true);
arg.setWsParameterOut(false);
getFactory.addWsSubprogFormal(arg);

//now we have to make stmts for if (this factory type) return this
for ( int i = 0; i < AFSubclasses.size(); i++)
{
    WsSelection ifElse = new WsSelection();

    WsExpression equals = new WsEqual();
    //left side variable name, factoryChoice
```

```
        ((WsEqual)equals).setWsBinExpOp1(new WsIdentifierRef(arg.getName()));
        //right side, value to match for factory
        ((WsEqual)equals).setWsBinExpOp2(new WsLiteralInteger(i+1));
        ifElse.setWsSelCondition(equals);

        WsAssignment bod = new WsAssignment();
        WsIdentifierRef t = new WsIdentifierRef(m.getName());
        t.setWsIdentRefTo(new WsIdentifier(identName));
        bod.setWsAssignLHS(t);
        WsExpression rhs = new WsAllocator();
        t = new WsIdentifierRef(AFSubclasses.get(i).getName());
        t.setWsIdentRefTo(new WsIdentifier(identName));
        ((WsAllocator)rhs).setWsAllocReturnType(t);
        bod.setWsAssignRHS(rhs);
        //only add if part, else is not used
        ifElse.addWsSelThenPart(bod);

        getFactory.addWsSubprogBody(ifElse);
    }
    theAFOps.add(m);

    //give the factory all its methods
    abstractFactory.setWsClassOperations(theAFOps);
    for ( int i = 0; i < AFSubclasses.size(); i++)
        target.addWsDecl(AFSubclasses.get(i));
    target.addWsDecl(abstractFactory);
    //set the factory's methods to what we made

    return true;
}
```

XformAF2 Transform

```
/**
 * Source file: XformAF2.java
 * Purpose: apply the usage of abstract factory pattern
 * <pre>
 * History:
 *    Original: 09-01-09 Gump created including methods:
 *              -XformAF2() default constructor
 *              -XformAF2(WsPackage) set up the ast
 *              -applicable(Object, OBject) check if can be applied
 *              -explain(Object) explain what will happen if can be applied
 *              -getDesc() description of transform
 *              -getName() name of xform
 *              -replay(WsPackage) not implemented
 *              -show() not implemented
 *              -toString() nothing really
 *              -undo(WsPackage, Object) not implemented
 *    Modified:
 * </pre>
 */

/**
    * Checks if abstract factory usage can be applied.  To be applied, the params
    * must be a list of lists, where the first item is an abstract super class
    * of some product and subsequent elements are concrete "themes" of that super
    * class object.  The final element in this list is the name of the recently
    * created abstract factor.  In addition, the abstract factory transform must have been
    * invoked prior to doing this transform.
    * @param tgt the ast to be modified
    * @param params the list of lists
    * @return whether or not the pattern can be applied
    */
   public static boolean applicable(Object tgt, Object params)
   {
      //must be a vector of vectors of class
      if ( !(params instanceof Vector))
         return false;
      Vector<Vector<WsClass>> classes = new Vector<Vector<WsClass>>();

      //should be vector of vectors classes, one less element cus last one is
      //the name of factory
      Vector vect;
      try
      {
         vect = (Vector)params;
```

```
        for ( int i = 0; i < vect.size() - 1; i++)
            classes.add((Vector<WsClass>)vect.get(i));
        }
        catch (Exception e)
        {
            return false;
        }

        //doesn't serve to transform empty set of classes
        if ( classes.size() == 0 || classes.get(0).size() == 0)
            return false;

        //must be the same number of products for each type
        for ( int i = 0; i < classes.size()-1; i++)
            if ( classes.get(i).size() != classes.get(i+1).size())
                return false;

        //must be abstract super class, products have it as parents
        for ( int i = 0; i < classes.size(); i++)
        {
            WsClass parent = classes.get(i).get(0);
            if ( !parent.getWsClassAbstract())
                return false;
            for ( int j = 1; j < classes.get(i).size(); j++)
            {
                WsClass child = classes.get(i).get(j);
                if ( child.getSuperclassName() == null ||
!(child.getSuperclassName().equals(parent.getName())))
                    return false;
            }
        }

        try
        {
            Vector<WsClass> factNameVect = (Vector<WsClass>)vect.get(vect.size() -1 );
            if ( !factNameVect.get(0).getName().startsWith("AbstractFactory"))
            {
                System.out.println("Expected the class that was abstract factory created by" +
                        "previous transform, not found.");
                return false;
            }
        }
        catch ( Exception e )
        {
            System.out.println("Expecte the name of abstract factory to be last element," +
                    " not found!");
```

```
      return false;
    }

    Vector<WsDeclaration> decls = ((WsPackage)tgt).getWsDecls();
    Vector<WsClass> theClasses = new Vector<WsClass>();
    //turn the classes from a list of lists into just a list of classes
    Vector<WsClass> classes2 = new Vector<WsClass>();
    for ( int i = 0; i < classes.size(); i++)
    {
      for ( int j = 0; j < classes.get(i).size(); j++)
      {
        classes2.add(classes.get(i).get(j));
      }
    }

    //only add classes that are not the factory super classes/subclasses
    //or factory itself or a concrete factory made
    for ( int i = 0; i < decls.size(); i++ )
    {
      if ( decls.get(i) instanceof WsClass && classNotInVector((WsClass)decls.get(i),
classes2)
          && !decls.get(i).getName().startsWith("ConcreteFactory")
          && !decls.get(i).getName().startsWith("AbstractFactory"))
        theClasses.add((WsClass)decls.get(i));

    }
    //now check every class
    for ( int h = 0; h < theClasses.size(); h++ )
    {
      Vector<WsMethod> meths = theClasses.get(h).getWsClassOperations();
      //System.err.println("Doing method evaluation for " +
theClasses.get(h).getName());
      for ( int i = 0; i < meths.size(); i++)
      {
        //get all the statements for this method
        Vector<WsStatement> stmts =
meths.get(i).getWsMethodSubprogram().getWsSubprogBody();
        allStmts = new Vector<WsStatement>();
        for ( int k = 0; k < stmts.size(); k++)
        {
          //only these two have to be searched recursively because
          //they could have the selects/iters in their body
          if ( stmts.get(k) instanceof WsSelection)
          {
            recursiveFindSelStmts((WsSelection)stmts.get(k));
          }
```

```
                else if ( stmts.get(k) instanceof WsIteration)
                {
                   recursiveFindIterStmts((WsIteration)stmts.get(k));
                }
                else
                   allStmts.add(stmts.get(k));
             }
             for ( int k = 0; k < allStmts.size(); k++)
             {
                //is the function being called in the super class?
                //we can't change the type of attributes to the super class
                //if they call non-inherited methods or we would have to type
                //cast
                if ( allStmts.get(k) instanceof WsProcedureCall)
                {
                   WsProcedureCall call = (WsProcedureCall)allStmts.get(k);
                   //we want the variable name before the "." notation
                   if ( call.getName().indexOf(".") == -1)
                      continue;//this wasn't a something.call()
                   String varName = call.getName().substring(0,
call.getName().indexOf("."));
                   WsObject var = null;
                  //is this a class attribute we are calling something on?
                   var = theClasses.get(h).getAttribute(varName);
                   //may be a function local variable otherwise
                   if ( var == null )
                   {
                      Vector<WsVariable> vars = meths.get(i).getFormals();
                      for ( int l = 0; l < vars.size(); l++ )
                         if ( vars.get(l).getName().equals(varName))
                            var = vars.get(l);
                   }
                   //is this a subclass of or type we are modding?
                   boolean subClassType = false;
                   //check each of the subclasses for if it is the same type
                   WsClass sup = new WsClass();
                   for ( int l = 0; l < classes2.size(); l++)
                   {
                      //System.err.println(classes.get(l).getName());
                      if ( var instanceof WsAttribute)
                      {
                         if ( (
((WsAttribute)var).getTypeName().equals(classes2.get(l).getName()) )
                         {
                            subClassType = true;
                            sup = (WsClass)XformUtils.getDeclByName((WsPackage)tgt,
```

```
classes2.get(l).getSuperclassName());
                        }
                        //System.err.println(((WsAttribute)var).getTypeName());
                    }
                    if ( var instanceof WsVariable)
                    {
                        if ( ((WsVariable)var).getTypeName().equals(classes2.get(l).getName()) )
                        {
                            subClassType = true;
                            sup = (WsClass)XformUtils.getDeclByName((WsPackage)tgt, classes2.get(l).getSuperclassName());
                        }
                        //System.err.println(((WsVariable)var).getTypeName());
                    }
                }
                //System.err.println(subClassType);
                String funcName = call.getName().substring(call.getName().indexOf(".")+1);
                //did we try to call a sub class funtion that is not in the super class??
                if ( subClassType && sup != null && sup.getWsClassOperation(funcName) == null )
                {
                    System.out.println("Tried to call " + call.getName() + " but" +
                            " it doesn't exist in parent class!\n  Can only" +
                            " update to use if only inherited methods are called");
                    return false;
                }
                //System.err.println(funcName);
            }
            //did we try to assign a value to an a variable calling a
            //parent class function?
            else if ( allStmts.get(k) instanceof WsAssignment )
            {
                WsAssignment asgn = (WsAssignment)allStmts.get(k);
                if ( asgn.getWsAssignRHS() instanceof WsFunctionCall)
                {
                    WsFunctionCall call = (WsFunctionCall)asgn.getWsAssignRHS();
                    //we want the variable name before the "." notation
                    String varName = call.getName().substring(0, call.getName().indexOf("."));
                    WsObject var = null;
                var = theClasses.get(h).getAttribute(varName);
                //may be a function local
                if ( var == null )
                {
```

```
                    Vector<WsVariable> vars = meths.get(i).getFormals();
                    for ( int l = 0; l < vars.size(); l++ )
                       if ( vars.get(l).getName().equals(varName))
                          var = vars.get(l);
                 }
                    boolean subClassType = false;
                    WsClass sup = new WsClass();
                    for ( int l = 0; l < classes2.size(); l++)
                    {
                       if ( var instanceof WsAttribute)
                       {
                          if                                                    (
((WsAttribute)var).getTypeName().equals(classes2.get(l).getName()) )
                          {
                             subClassType = true;
                             sup   =   (WsClass)XformUtils.getDeclByName((WsPackage)tgt,
classes2.get(l).getSuperclassName());
                          }
                       }
                       if ( var instanceof WsVariable)
                       {
                          if                                                    (
((WsVariable)var).getTypeName().equals(classes2.get(l).getName()) )
                          {
                             subClassType = true;
                             sup   =   (WsClass)XformUtils.getDeclByName((WsPackage)tgt,
classes2.get(l).getSuperclassName());
                          }
                       }
                    }
                    String                                  funcName                                =
call.getName().substring(call.getName().indexOf(".")+1);
                    if ( subClassType && sup.getWsClassOperation(funcName) == null )
                    {
                       System.out.println("Tried to call " + call.getName() + " but" +
                          " it doesn't exist in parent class!\n  Can only" +
                          " update to use if only inherited methods are called");
                       return false;
                    }
                    //System.err.println(call.getName());
                 }
              }
           }
        }
     }
     return true;
```

```
}

/**
 * Returns true if the class specified is not in the vector of clases passed
 * as second argument.
 * @param find a class to search for in second list
 * @param searchList a list of classes to search through
 * @return true if the passed class was not in the vector of classes
 */
private static boolean classNotInVector(WsClass find, Vector<WsClass> searchList)
{
   for ( int i = 0; i < searchList.size(); i++)
   {
      if ( find.getName().equals(searchList.get(i).getName()))
         return false;
   }
   return true;
}

/**
 * Applies the abstract factory to the existing model passed into constructor.
 * Does so by creating an abstract factory super class and concrete subclasses
 * that can return types of related objects as specified by the params.
 * @param params a list of lists, where the sub list have the first item as
 * an abstract product super class and the subsequent elements are subclasses
 * of the super class representing a "theme" of that object.
 * @return success or failure
 */
@Override
public boolean execute(Object params)
{
   //don't bother trying to apply if not applicable
   if ( !applicable(target, params))
      return false;

   Vector<Vector<WsClass>> listClasses = (Vector<Vector<WsClass>>)params;



   //the classes were a list of lists, but less complicated checking if
   //we just turn them back into a flat list
   /*Vector<WsClass> flatClasses = new Vector<WsClass>();
   for ( int i = 0; i < vect.size() - 1; i++)
      for ( int j = 0; j < ((Vector<WsClass>)vect.get(i)).size(); j++)
         flatClasses.add(((Vector<WsClass>)vect.get(i)).get(j));*/
```

```
        //last element is the factory created previously as a single class
        WsClass factory = listClasses.get(listClasses.size()-
1).get(0);//Vector<WsClass>)vect.get(vect.size()-1)).get(0);


        //update each of the classes

        Vector<WsClass> theClasses = new Vector<WsClass>();
        Vector<WsDeclaration> allDecls = target.getWsDecls();

        //go through all the declarations and find the newly created factories
        //this relies on the fact that all the newly created factories are the
        //last declarations made so searching in reverse order will find these
        //factories first
        Vector<WsClass> theFactories = new Vector<WsClass>();
        //for ( int i = 0; i < listClasses.size(); i++)
         //   theFactories.add(listClasses.get(i).get(0));
        int found = 0;
        for ( int i = (allDecls.size() - 1); i >= 0 && found != (listClasses.get(0).size() - 1); i--
)
           if ( allDecls.get(i) instanceof WsClass)
              if ( ((WsClass)allDecls.get(i)).getName().startsWith("ConcreteFactory"))
              {
                 theFactories.insertElementAt((WsClass)allDecls.get(i), 0);
                 found++;
              }

        //go through all the declarations and find the classes
        //who are not in the set of factory classes so that we can update
        //the usage in them
        int count = 0;
        for ( int i = 0; i < allDecls.size(); i++)
        {
           count = 0;
           if ( allDecls.get(i) instanceof WsClass)
              for ( int j = 0; j < listClasses.size(); j++)
              {
                 //we count number of times it wasn't in there
                 //it must not be in any of the lists to be added right
                 if ( classNotInVector( (WsClass)allDecls.get(i), listClasses.get(j))
                 && classNotInVector((WsClass)allDecls.get(i), theFactories))
                    count++;
                 if ( count == listClasses.size())
                    theClasses.add((WsClass)allDecls.get(i));
              }
```

```
    }


    for ( int i = 0; i < theClasses.size(); i++ )
    {
       Vector<WsMethod> meths = theClasses.get(i).getWsClassOperations();
       //this flag tells whether a method used subclass
       boolean usedASubClass = false;
       //flag to tell if any method used the subclass to add attribute to class
       boolean anyClassUsed = false;
       //check each method
       for ( int j = 0; j < meths.size(); j++ )
       {
          Vector<WsStatement> stmts =
meths.get(j).getWsMethodSubprogram().getWsSubprogBody();
          allStmts = new Vector<WsStatement>();
          //check all the statements
          for ( int k = 0; k < stmts.size(); k++ )
          {
            //only these two have to be searched recursively because
            //they could have the selects/iters in their body
            if ( stmts.get(k) instanceof WsSelection)
            {
              recursiveFindSelStmts((WsSelection)stmts.get(k));
            }
            else if ( stmts.get(k) instanceof WsIteration)
            {
               recursiveFindIterStmts((WsIteration)stmts.get(k));
            }
            else
               allStmts.add(stmts.get(k));
          }
          //System.err.println("The function " + meths.get(j).getName() + " has " +
allStmts.size());
          for ( int k = 0; k < allStmts.size(); k++)
          {
            if ( allStmts.get(k) instanceof WsAssignment)
            {
              WsAssignment s = (WsAssignment)allStmts.get(k);
              //is this something = new somethingElse?
              if ( s.getWsAssignRHS() instanceof WsAllocator)
              {
                 WsAllocator alloc = (WsAllocator)s.getWsAssignRHS();
                 //is this an instance of our class?
                 for ( int l = 0; l < listClasses.size(); l++)
                 {
```

```
                    Vector<WsClass> productFamily = listClasses.get(l);
                    //m = 1 because the first is abstract super class
                    //we don't care about that
                    for ( int n = 1; n < productFamily.size(); n++)
                    {
                        if                                                                    (
alloc.getWsAllocReturnType().getName().equals(productFamily.get(n).getName()))
                        {
                           System.err.println(alloc.getWsAllocReturnType().getName() + "
AND " + productFamily.get(n).getName());
                        //replace it with our function call instead!
                        WsFunctionCall callFact = new WsFunctionCall();

callFact.setWsSubprogCallName("the"+factory.getName()+".create"+productFamily.get(
n).getSuperclassName());
                        s.setWsAssignRHS(callFact);

                        //right before we can do this assignment, make
                        //sure we shift to right factory type
                        WsAssignment getFact = new WsAssignment();
                        getFact.setWsAssignLHS(new
WsIdentifierRef("the"+factory.getName()));

                        WsFunctionCall call = new WsFunctionCall();
                        call.setWsSubprogCallName(factory.getName()+".getFactory");
                        Vector<WsExpression> args = new Vector<WsExpression>();
                        args.add(new WsLiteralInteger(n));
                        call.setWsSubprogCallArgs(args);
                        getFact.setWsAssignRHS(call);
                        if (!usedASubClass)
                            stmts.insertElementAt(getFact, 0);

                        usedASubClass = true; anyClassUsed = true;

                        //change the attribute type to product class type
                        WsName lhsName = s.getWsAssignLHS();
                        Vector<WsVariable> vars = meths.get(j).getFormals();
                        boolean attr = false;
                        for ( int m = 0; m < vars.size(); m++ )
                        {
                           if ( vars.get(m).getName().equals(lhsName.getName()))

vars.get(m).setTypeName(productFamily.get(n).getSuperclassName());
                           attr = true;
                        }
                        if ( !attr && theClasses.get(i).getAttribute(lhsName.getName()) !=
```

```
null )
                        {

theClasses.get(i).getAttribute(lhsName.getName()).setTypeName(productFamily.get(n).g
etSuperclassName());
                        }

                    }
                  }
                }

              }
            }

           }
         }
         //we made it through a class, did it actually need to use the factory or no?
         if ( anyClassUsed )
         {
            WsAttribute theFact = new WsAttribute();
            theFact.setWsPrivate(true);
            theFact.setWsStatic(false);
            theFact.setName("the"+factory.getName());
            theFact.setTypeName(factory.getName());
            theClasses.get(i).addAttribute(theFact);
         }
      }

      return true;
   }

/**
    * Recursively explores if/elses to fine any nested statements
    * @param wsSelection initial if/else to search
    */
    private static void recursiveFindSelStmts(WsSelection wsSelection)
   {
      Vector<WsStatement> stmtsE = wsSelection.getWsSelElsePart();
      Vector<WsStatement> stmtsT = wsSelection.getWsSelThenPart();
      //WsStatement blank = new WsAssignment();
      for ( int i = 0; i < stmtsE.size(); i++)
      {
         if ( stmtsE.get(i) instanceof WsSelection)
         {
            //allStmts.add(blank);
            recursiveFindSelStmts((WsSelection)stmtsE.get(i));
```

```
        }
        else if ( stmtsE.get(i) instanceof WsIteration)
        {
            //allStmts.add(blank);
            recursiveFindIterStmts((WsIteration)stmtsE.get(i));
        }
        else
            allStmts.add(stmtsE.get(i));
    }
    for ( int i = 0; i < stmtsT.size(); i++)
    {
        if ( stmtsT.get(i) instanceof WsSelection)
        {
            //allStmts.add(blank);
            recursiveFindSelStmts((WsSelection)stmtsT.get(i));
        }
        else if ( stmtsT.get(i) instanceof WsIteration)
        {
            //allStmts.add(blank);
            recursiveFindIterStmts((WsIteration)stmtsT.get(i));
        }
        else
            allStmts.add(stmtsT.get(i));
    }
}

/**
 * Recursively explores while loops to fine any nested statements
 * @param wsSelection initial while loop to search
 */
private static void recursiveFindIterStmts(WsIteration wsIteration)
{
    Vector<WsStatement> stmtsE = wsIteration.getWsIterBody();
    //WsStatement blank = new WsAssignment();
    for ( int i = 0; i < stmtsE.size(); i++)
    {
        if ( stmtsE.get(i) instanceof WsSelection)
        {
            //allStmts.add(blank);
            recursiveFindSelStmts((WsSelection)stmtsE.get(i));
        }
        else if ( stmtsE.get(i) instanceof WsIteration)
        {
            //allStmts.add(blank);
            recursiveFindIterStmts((WsIteration)stmtsE.get(i));
        }
```

```
        else
            allStmts.add(stmtsE.get(i));
    }
}
```

XformFactM1 Transform

```
/**
 * Source file: XformFactM1.java
 * Purpose: apply the basic factory methods pattern transform to existing model.
 * <pre>
 * History:
 *    Original: 07-18-09 Gump created including methods:
 *              -XformFactM1() default constructor
 *              -XformFactM1(WsPackage) set up the ast
 *              -applicable(Object, OBject) check if can be applied
 *              -explain(Object) explain what will happen if can be applied
 *              -getDesc() description of transform
 *              -getName() name of xform
 *              -replay(WsPackage) not implemented
 *              -show() not implemented
 *              -toString() nothing really
 *              -undo(WsPackage, Object) not implemented
 *    Modified:
 * </pre>
 */

/**
    * Checks if FactM1 can be applied.  To be applied, a list of class names
    * must be passed in where each class has a common super class and te class
    * named SuperClassNameFactory must not already exist in the model.  Also,
    * the specified classes must have a default constructor even if it does nothing.
    * @param tgt the ast to be modified
    * @param params vector of class names that inherit from a common super class
    * @return whether or not the pattern can be applied
    */
   public static boolean applicable(Object tgt, Object params)
   {
      if ( tgt == null )
         return false;
      if ( !(params instanceof Vector))
      {
         System.out.println("Expected vector, not found.");
         return false;
      }

      Vector<WsClass> classes = new Vector<WsClass>();
      Vector<String> sClasses;
      //forcibly try to convert to strings to see if they are
      try
      {
```

```
            sClasses = (Vector<String>)params;
        }
        catch (Exception e)
        {
            System.out.println("Expected string class names.");
            return false;
        }

        if ( sClasses.size() == 0 )
        {
            System.out.println("Will not perform transform with no classes.");
            return false;
        }

        //forcibly try to convert to classes to see if they are
        try
        {
            for ( int i = 0; i < sClasses.size(); i++)
                classes.add((WsClass)XformUtils.getDeclByName((WsPackage)tgt,
sClasses.get(i)));
        }
        catch (Exception e)
        {
            System.out.println("Expected list containing all classes, but did not find one.");
            return false;
        }

        //check for common super class
        String sc = classes.get(0).getSuperclassName();
        if ( sc == null || sc.trim().equals(""))
        {
            System.out.println("Expected super class, found none.");
            return false;
        }
        for ( int i = 1; i < classes.size(); i++)
            if ( !classes.get(i).getSuperclassName().equals(sc) )
            {
                System.out.println("Expected common super class, not found.");
                return false;
            }

       //does the super class exist
        try
        {
            WsClass sup = (WsClass)XformUtils.getDeclByName((WsPackage)tgt, sc);
        }
```

```
    catch (Exception e)
    {
       System.out.println("Couldn't find the super class in the model, required!");
       return false;
    }

    //can't already be a class named <superclass>Factory
    if ( XformUtils.getDeclByName((WsPackage)tgt, sc+"Factory") != null )
    {
       System.out.println("Found " + sc + "Factory, was pattern already applied?");
       return false;
    }
    //classes must have default constructor
    for ( int i = 0; i < classes.size(); i++ )
    {
       Vector<WsMethod> methods = classes.get(i).getWsClassOperations();
       boolean defaultCons = false;
       for ( int k = 0; k < methods.size(); k++)
       {
          WsMethod meth = methods.get(k);
          //must be a constructor, so same name as class
          if ( meth.getName().equals(classes.get(i).getName()))
          {
             //must have no args
             if ( meth.getFormals() != null || meth.getFormals().size() != 0)
             {
                defaultCons = true;
             }
          }
          if ( !defaultCons )
          {
             System.out.println("No default constructor found, expected one.");
             return false;
          }
       }
    }
    return true;
}


/**
 * Applies the FactM1 to the existing model passed into constructor.  This will
 * happen by creating a new class named SuperClassNameFactory (where super class
 * name is super class of the classes supplied) with necessary logic to
 * return a correct instance of a logger based on criteria.
 * @param params a vector or subclass string names with a common super class
```

```
 * @return success or failure
 */
@Override
public boolean execute(Object params)
{
   if ( !applicable(target, params) )
      return false;

   Vector<WsClass> classes = new Vector<WsClass>();
   Vector<String> sClasses;
   //forcibly convert to strings to see if they are
   sClasses = (Vector<String>)params;


   //forcibly convert to classes
   for ( int i = 0; i < sClasses.size(); i++)
      classes.add((WsClass)XformUtils.getDeclByName(target, sClasses.get(i)));

   //get super class name
   String sc = classes.get(0).getSuperclassName();


   WsClass fact = new WsClass();
   fact.setName(sc+"Factory");
   fact.setWsClassAbstract(false);

   //create default constructor action!  Borrowed from Arjun1!
   WsFunction constructor = new WsFunction();
   WsMethod mC = new WsMethod(constructor);
   mC.setWsPrivate(false);
   mC.setWsClassMethod(true);
   constructor.setName(sc+"Factory");
   fact.addWsClassOperation(mC);

   WsIdentifier identC = fact.getWsDeclName();
   String identNameC = identC.getWsIdentSymbol();
   WsIdentifierRef returnIdentC = new WsIdentifierRef(identNameC);
   returnIdentC.setWsIdentRefTo(identC);
   constructor.setWsFunctReturnType(returnIdentC);
   //end borrowed part!!!!!!!!!!

   //make storage to decide which type for factory to return
   Vector<WsAttribute> flags = new Vector<WsAttribute>();
   for ( int i = 0; i < sClasses.size(); i++)
   {
      WsAttribute tmp = new WsAttribute();
```

```
    tmp.setName(sClasses.get(i)+"Flag");
    tmp.setAttributeTypeName("Boolean");
    tmp.setWsPrivate(true);
    tmp.setWsStatic(false);
    flags.add(tmp);
}
fact.setWsClassDataComponents(flags);

//create get instance method
WsFunction getF = new WsFunction();
getF.setName("getInstance");
WsIdentifierRef returnIdent = new WsIdentifierRef(sc);
returnIdent.setWsIdentRefTo(new WsIdentifier(sc));
getF.setWsFunctReturnType(returnIdent);
getF.setWsName(new WsIdentifier("getInstance"));
WsMethod m = new WsMethod(getF);
m.setWsAbstract(false);
m.setWsPrivate(false);
m.setFunctReturnType(sc);
//set up if logic to return right kind of factory
WsVariable type = new WsVariable();
type.setName("tmp");
type.setTypeName(sc);
getF.addWsSubprogLocal(type);

//makes statements of the form
//if ( flag == true)
//tmp = new Object();
for ( int i = 0; i < sClasses.size(); i++)
{
    WsSelection ifElse = new WsSelection();
    //if classFlag == true
    WsExpression cond = new WsEqual();
    WsIdentifierRef t = new WsIdentifierRef(sClasses.get(i)+"Flag");
    t.setWsIdentRefTo(new WsIdentifier(sc));
    ((WsEqual)cond).setWsBinExpOp1(t);
    t = new WsIdentifierRef("true");
    t.setWsIdentRefTo(new WsIdentifier("Boolean"));
    ((WsEqual)cond).setWsBinExpOp2(t);
    ifElse.setWsSelCondition(cond);
    WsAssignment bod = new WsAssignment();
    bod.setWsAssignLHS(new WsIdentifierRef(type.getName()));
    WsExpression rhs = new WsAllocator();
    t = new WsIdentifierRef(sClasses.get(i));
    t.setWsIdentRefTo(new WsIdentifier(sClasses.get(i)));
    ((WsAllocator)rhs).setWsAllocReturnType(t);
```

```
            bod.setWsAssignRHS(rhs);
            ifElse.addWsSelThenPart(bod);
            getF.addWsSubprogBody(ifElse);
        }
        //return tmp
        WsAssignment bod = new WsAssignment();
        bod.setWsAssignLHS(new WsIdentifierRef("getInstance"));
        WsExpression rhs = new WsIdentifierRef(type.getName());
        //((WsAllocator)rhs).setWsAllocReturnType(new
WsIdentifierRef(type.getName()));
        bod.setWsAssignRHS(rhs);
        getF.addWsSubprogBody(bod);
        fact.addWsClassOperation(m);

        //set up sets to initialize the correct factory to be used
        for ( int i = 0; i < sClasses.size(); i++)
        {
            //create method to useClassName
            WsProcedure setter = new WsProcedure();
            setter.setName("use"+sClasses.get(i));
            m = new WsMethod(setter);
            m.setWsPrivate(false);
            setter.setWsName(new WsIdentifier("use"+sClasses.get(i)));

            //series of statements to set up boolean values correcty
            for ( int j = 0; j < classes.size(); j++)
            {
                WsAssignment set = new WsAssignment();
                set.setWsAssignLHS(new WsIdentifierRef(sClasses.get(i)+"Flag"));
                if ( i != j ) //don't use this oject type so set to false
                {
                    set.setWsAssignRHS(new WsIdentifierRef("false"));
                }
                else //use this object type so set to true
                {
                    set.setWsAssignRHS(new WsIdentifierRef("true"));
                }
                setter.addWsSubprogBody(set);
            }
            fact.addWsClassOperation(m);
        }

        target.addWsDecl(fact);
        return true;
    }
```

XformFactM2 Transform

```
/**
 * Source file: XformFactM2.java
 * Purpose: update the model to use the applied factor method model
 * <pre>
 * History:
 *    Original: 08-08-09 Gump created including methods:
 *              -XformFactM2() default constructor
 *              -XformFactM2(WsPackage) set up the ast
 *              -applicable(Object, OBject) check if can be applied
 *              -explain(Object) explain what will happen if can be applied
 *              -getDesc() description of transform
 *              -getName() name of xform
 *              -replay(WsPackage) not implemented
 *              -show() not implemented
 *              -toString() nothing really
 *              -undo(WsPackage, Object) not implemented
 *    Modified:
 * </pre>
 */

/**
    * Checks if FactM2 can be applied.  To be applied, the class specifed as
    * an argument must have already been transformmed into a factory method.  Also
    * any calls to sub class methods must be in the parent class as well otherwise
    * calling would require typecasting after this transform.
    * @param tgt the ast to be modified
    * @param params the classes that inherit from a common super class
    * that we created a factory method for
    * @return whether or not the pattern can be applied
    */
   public static boolean applicable(Object tgt, Object params)
   {
      if ( tgt == null )
         return false;
       if ( !(params instanceof Vector))
      {
         System.out.println("Expected vector, not found.");
         return false;
      }

      Vector<WsClass> classes = new Vector<WsClass>();
      Vector<String> sClasses;
      //forcibly try to convert to strings to see if they are
      try
```

```
        {
            sClasses = (Vector<String>)params;
        }
        catch (Exception e)
        {
            System.out.println("Expected string class names.");
            return false;
        }

        if ( sClasses.size() == 0 )
        {
            System.out.println("Will not perform transform with no classes.");
            return false;
        }

        //forcibly try to convert to classes to see if they are
        try
        {
            for ( int i = 0; i < sClasses.size(); i++)
                classes.add((WsClass)XformUtils.getDeclByName((WsPackage)tgt,
sClasses.get(i)));
        }
        catch (Exception e)
        {
            System.out.println("Expected list containing all classes, but did not find one.");
            return false;
        }

        //check for common super class
        String sc = classes.get(0).getSuperclassName();
        if ( sc == null || sc.trim().equals(""))
        {
            System.out.println("Expected super class, found none.");
            return false;
        }
        for ( int i = 1; i < classes.size(); i++)
            if ( !classes.get(i).getSuperclassName().equals(sc) )
            {
                System.out.println("Expected common super class, not found.");
                return false;
            }

        //does the super class exist
        try
        {
            WsClass sup = (WsClass)XformUtils.getDeclByName((WsPackage)tgt, sc);
```

```
            if ( sup == null)
               throw new Exception();
         }
         catch (Exception e)
         {
            System.out.println("Couldn't find the super class in the model, required!");
            return false;
         }

         //see if SuperClassNameFactor exists already
         if ( XformUtils.getDeclByName((WsPackage)tgt, sc + "Factory") == null )
         {
            System.out.println("Was expecting class  " + sc + "Factory." +
                  "  Was previous pattern not already applied?");
            return false;
         }

         //gather all the statements
         WsClass sup = (WsClass)XformUtils.getDeclByName((WsPackage)tgt, sc);
         Vector<WsDeclaration> decls = ((WsPackage)tgt).getWsDecls();
         Vector<WsClass> theClasses = new Vector<WsClass>();
         for ( int i = 0; i < decls.size(); i++ )
         {
            //only add classes that aren't the super class or parent class since
            //they wouldnt be using each other
            if ( decls.get(i) instanceof WsClass &&
!((WsClass)decls.get(i)).getName().equals(sc) )
            {
               int count = 0;
               for ( int j = 0; j < classes.size(); j++)
               {
                  if ( !((WsClass)decls.get(i)).getName().equals(classes.get(j).getName()) )
                     count++;
               }
               //this way we will only check things that are not parent/subclass of the factory
               if ( count == classes.size() )
                  theClasses.add((WsClass)decls.get(i));
            }
         }
         //now check every class
         for ( int h = 0; h < theClasses.size(); h++ )
         {
            Vector<WsMethod> meths = theClasses.get(h).getWsClassOperations();
            //System.err.println("Doing method evaluation for " +
theClasses.get(h).getName());
            for ( int i = 0; i < meths.size(); i++)
```

```
        {
          //get all the statements for this method
          Vector<WsStatement> stmts =
meths.get(i).getWsMethodSubprogram().getWsSubprogBody();
          allStmts = new Vector<WsStatement>();
          for ( int k = 0; k < stmts.size(); k++)
          {
              //only these two have to be searched recursively because
              //they could have the selects/iters in their body
              if ( stmts.get(k) instanceof WsSelection)
              {
                recursiveFindSelStmts((WsSelection)stmts.get(k));
              }
              else if ( stmts.get(k) instanceof WsIteration)
              {
                 recursiveFindIterStmts((WsIteration)stmts.get(k));
              }
              else
                 allStmts.add(stmts.get(k));
          }
          for ( int k = 0; k < allStmts.size(); k++)
          {
             //is the function being called in the super class?
             //we can't change the type of attributes to the super class
             //if they call non-inherited methods or we would have to type
             //cast
             if ( allStmts.get(k) instanceof WsProcedureCall)
             {
                WsProcedureCall call = (WsProcedureCall)allStmts.get(k);
                //we want the variable name before the "." notation
                if ( call.getName().indexOf(".") == -1)
                   continue;//this wasn't a something.call()
                String varName = call.getName().substring(0,
call.getName().indexOf("."));
                WsObject var = null;
               //is this a class attribute we are calling something on?
                var = theClasses.get(h).getAttribute(varName);
                //may be a function local variable otherwise
                if ( var == null )
                {
                   Vector<WsVariable> vars = meths.get(i).getFormals();
                   for ( int l = 0; l < vars.size(); l++ )
                      if ( vars.get(l).getName().equals(varName))
                         var = vars.get(l);
                }
                //is this a subclass of or type we are modding?
```

```
                boolean subClassType = false;
                //check each of the subclasses for if it is the same type
                for ( int l = 0; l < classes.size(); l++)
                {
                   //System.err.println(classes.get(l).getName());
                   if ( var instanceof WsAttribute)
                   {
                      if ( ((WsAttribute)var).getTypeName().equals(classes.get(l).getName()) )
                         subClassType = true;
                      //System.err.println(((WsAttribute)var).getTypeName());
                   }
                   if ( var instanceof WsVariable)
                   {
                      if ( ((WsVariable)var).getTypeName().equals(classes.get(l).getName()) )
                         subClassType = true;
                      //System.err.println(((WsVariable)var).getTypeName());
                   }
                }
                //System.err.println(subClassType);
                String funcName = call.getName().substring(call.getName().indexOf(".")+1);
                //did we try to call a sub class funtion that is not in the super class??
                if ( subClassType && sup != null && sup.getWsClassOperation(funcName) == null )
                {
                   System.out.println("Tried to call " + call.getName() + " but" +
                         " it doesn't exist in parent class!\n  Can only" +
                         " update to use if only inherited methods are called");
                   return false;
                }
                //System.err.println(funcName);
             }
             //did we try to assign a value to an a variable calling a
             //parent class function?
             else if ( allStmts.get(k) instanceof WsAssignment )
             {
                WsAssignment asgn = (WsAssignment)allStmts.get(k);
                if ( asgn.getWsAssignRHS() instanceof WsFunctionCall)
                {
                   WsFunctionCall call = (WsFunctionCall)asgn.getWsAssignRHS();
                   //we want the variable name before the "." notation
                   String varName = call.getName().substring(0, call.getName().indexOf("."));
                   WsObject var = null;
```

```
                var = theClasses.get(h).getAttribute(varName);
                //may be a function local
                if ( var == null )
                {
                   Vector<WsVariable> vars = meths.get(i).getFormals();
                   for ( int l = 0; l < vars.size(); l++ )
                      if ( vars.get(l).getName().equals(varName))
                         var = vars.get(l);
                }
                   boolean subClassType = false;
                   for ( int l = 0; l < classes.size(); l++)
                   {
                      if ( var instanceof WsAttribute)
                      {
                         if (
((WsAttribute)var).getTypeName().equals(classes.get(l).getName()) )
                            subClassType = true;
                      }
                      if ( var instanceof WsVariable)
                      {
                         if (
((WsVariable)var).getTypeName().equals(classes.get(l).getName()) )
                            subClassType = true;
                      }
                   }
                   String                                funcName                                =
call.getName().substring(call.getName().indexOf(".")+1);
                   if ( subClassType && sup.getWsClassOperation(funcName) == null )
                   {
                      System.out.println("Tried to call " + call.getName() + " but" +
                         " it doesn't exist in parent class!\n  Can only" +
                         " update to use if only inherited methods are called");
                      return false;
                   }
                   //System.err.println(call.getName());
                }
             }
          }
       }

       //check if classes have an attribute named "the<superClassName>Factory"
       //because it will be created to get instances of factory
       //they also must have a default constructor to instansiate it in
       //classes must have default constructor
       for ( int i = 0; i < theClasses.size(); i++ )
```

```
    {
      Vector<WsMethod> methods = theClasses.get(i).getWsClassOperations();
      boolean defaultCons = false;
      for ( int k = 0; k < methods.size(); k++)
      {
        WsMethod meth = methods.get(k);
        //must be a constructor, so same name as class
        if ( meth.getName().equals(theClasses.get(i).getName()))
        {
          //must have no args
          if ( meth.getFormals() != null || meth.getFormals().size() != 0)
          {
            defaultCons = true;
          }
        }
        //abstract classes wouldn't need a constructor now would they?
        if ( !defaultCons && !theClasses.get(i).getWsClassAbstract())
        {
          System.out.println("No default constructor found, expected one for " +
              theClasses.get(i).getName() + ".");
          return false;
        }
      }
      if ( theClasses.get(i).getAttribute("the"+sc+"Factory") != null )
      {
        System.out.println("Expected not to have an attribute named the" + sc +
"Factory, " +
            "but found one!");
        return false;
      }
    }
    return true;
  }

  /**
   * Applies the FactM2 to the existing model passed into constructor.
   * Does so by updating all instansiating of the class with calls
   * to the factory's method that tells it which type of subclass it should return
   * as well as replace instantiations with calls to the getInstancce().  Also
   * adds an attribute of the type of factory to classes that use it.
   * @param params the class to be transformed to use FactM1
   * @return success or failure
   */
  @Override
  public boolean execute(Object params)
  {
```

```
        if ( !applicable(target, params) )
           return false;

        //turn into a class because it should be!
        Vector<WsClass> classes = new Vector<WsClass>();
        Vector<String> sClasses;
        //forcibly try to convert to strings
        sClasses = (Vector<String>)params;

        //forcibly try to convert to classes
        for ( int i = 0; i < sClasses.size(); i++)
           classes.add((WsClass)XformUtils.getDeclByName(target, sClasses.get(i)));
        String sc = classes.get(0).getSuperclassName();
        //get all classes in the model
        Vector<WsDeclaration> decls = target.getWsDecls();
        Vector<WsClass> theClasses = new Vector<WsClass>();
        for ( int i = 0; i < decls.size(); i++ )
        {
           //only add classes that aren't the super class or parent class since
           //they wouldnt be using each other
           if ( decls.get(i) instanceof WsClass &&
!((WsClass)decls.get(i)).getName().equals(sc) &&
                 !((WsClass)decls.get(i)).getName().equals(sc+"Factory"))
           {
              int count = 0;
              for ( int j = 0; j < classes.size(); j++)
              {
                 if ( !((WsClass)decls.get(i)).getName().equals(classes.get(j).getName()) )
                    count++;
              }
              //this way we will only check things that are not parent/subclass of the factory
              if ( count == classes.size() )
                 theClasses.add((WsClass)decls.get(i));
           }
        }

        WsFunctionCall getFact = new WsFunctionCall("the" + sc + "Factory.getInstance");
        //update each of the classes
        for ( int i = 0; i < theClasses.size(); i++ )
        {
           Vector<WsMethod> meths = theClasses.get(i).getWsClassOperations();
           //this flag tells whether or not we need to add a factory method attribute
           boolean usedASubClass = false;
           //check each method
           for ( int j = 0; j < meths.size(); j++ )
           {
```

```
            boolean methUsedSub = false;
            Vector<WsStatement>                                    stmts                                    =
meths.get(j).getWsMethodSubprogram().getWsSubprogBody();
            allStmts = new Vector<WsStatement>();
            //check all the statements
            for ( int k = 0; k < stmts.size(); k++ )
            {
               //only these two have to be searched recursively because
               //they could have the selects/iters in their body
               if ( stmts.get(k) instanceof WsSelection)
               {
                 recursiveFindSelStmts((WsSelection)stmts.get(k));
               }
               else if ( stmts.get(k) instanceof WsIteration)
               {
                  recursiveFindIterStmts((WsIteration)stmts.get(k));
               }
               else
                  allStmts.add(stmts.get(k));
            }
            System.err.println("The  function  " + meths.get(j).getName() + " has " +
allStmts.size());
            for ( int k = 0; k < allStmts.size(); k++)
            {
               if ( allStmts.get(k) instanceof WsAssignment)
               {
                 WsAssignment s = (WsAssignment)allStmts.get(k);
                 //is this something = new somethingElse?
                 if ( s.getWsAssignRHS() instanceof WsAllocator)
                 {
                    WsAllocator alloc = (WsAllocator)s.getWsAssignRHS();
                    //is this an instance of ou class?
                    for ( int l = 0; l < classes.size(); l++)
                    {
                       if                                                                                  (
alloc.getWsAllocReturnType().getName().equals(classes.get(l).getName()))
                       {
                          //replace it with our function call instead!
                          s.setWsAssignRHS(getFact);
                          //right before we can do this assignment, make
                          //sure we shift to right factory type
                          WsProcedureCall call = new WsProcedureCall();

call.setWsSubprogCallName("the"+sc+"Factory.use"+classes.get(l).getName());
                          if (!methUsedSub)
                              stmts.insertElementAt(call, 0);
```

```
                        usedASubClass = true; methUsedSub = true;
                        //need to set up factory right before we do assignment
                        /*for ( int m = 0; m < stmts.size(); m++ )
                        {
                           if ( stmts.get(m) == s)
                           {
                               stmts.insertElementAt(call, m);
                               break;
                           }
                           else if ( stmts.get(m) instanceof WsSelection )
                           {
                              found = false;
                              sG = s;
                              callG = call;
                              addUsageInSelection((WsSelection)stmts.get(m));
                           }
                           else if ( stmts.get(m) instanceof WsIteration )
                           {
                              found = false;
                              addUsageInIteration((WsIteration)stmts.get(m));
                           }
                        }*/

                        //change the attribute type to super class type
                        WsName lhsName = s.getWsAssignLHS();
                        Vector<WsVariable> vars = meths.get(j).getFormals();
                        boolean attr = false;
                        for ( int m = 0; m < vars.size(); m++ )
                        {
                           if ( vars.get(m).getName().equals(lhsName.getName()))
                              vars.get(m).setTypeName(sc);
                           attr = true;
                        }
                        if ( !attr && theClasses.get(i).getAttribute(lhsName.getName()) !=
null )
                        {

theClasses.get(i).getAttribute(lhsName.getName()).setTypeName(sc);
                        }

                     }
                  }
               }

            }
         }
```

```
            }
            //we made it through a class, did it actually need to use the factory or no?
            if ( usedASubClass )
            {
               WsAttribute theFact = new WsAttribute();
               theFact.setWsPrivate(true);
               theFact.setWsStatic(false);
               theFact.setName("the"+sc+"Factory");
               theFact.setTypeName(sc+"Factory");
               theClasses.get(i).addAttribute(theFact);

               //find the default constructor
               for ( int m = 0; m < meths.size(); m++)
               {
                  //no arguments and same name as class
                  if ( meths.get(m).getName().equals(theClasses.get(i).getName()) &&
                        (meths.get(m).getFormals() == null || meths.get(m).getFormals().size()
== 0) )
                  {
                     WsAssignment initFact = new WsAssignment();
                     initFact.setWsAssignLHS(new WsIdentifierRef("the"+sc+"Factory"));
                     WsExpression init = new WsAllocator();
                     WsIdentifierRef t = new WsIdentifierRef(sc+"Factory");
                     t.setWsIdentRefTo(new WsIdentifier(sc+"Factory"));
                     ((WsAllocator)init).setWsAllocReturnType(t);
                     initFact.setWsAssignRHS(init);

meths.get(m).getWsMethodSubprogram().addWsSubprogStatement(initFact);
                  }
               }
            }
         }

         return true;
      }

/**
    * Recursively explores if/elses to fine any nested statements
    * @param wsSelection initial if/else to search
    */
   private static void recursiveFindSelStmts(WsSelection wsSelection)
   {
      Vector<WsStatement> stmtsE = wsSelection.getWsSelElsePart();
      Vector<WsStatement> stmtsT = wsSelection.getWsSelThenPart();
      //WsStatement blank = new WsAssignment();
```

```
    for ( int i = 0; i < stmtsE.size(); i++)
    {
        if ( stmtsE.get(i) instanceof WsSelection)
        {
            //allStmts.add(blank);
            recursiveFindSelStmts((WsSelection)stmtsE.get(i));
        }
        else if ( stmtsE.get(i) instanceof WsIteration)
        {
            //allStmts.add(blank);
            recursiveFindIterStmts((WsIteration)stmtsE.get(i));
        }
        else
            allStmts.add(stmtsE.get(i));
    }
    for ( int i = 0; i < stmtsT.size(); i++)
    {
        if ( stmtsT.get(i) instanceof WsSelection)
        {
            //allStmts.add(blank);
            recursiveFindSelStmts((WsSelection)stmtsT.get(i));
        }
        else if ( stmtsT.get(i) instanceof WsIteration)
        {
            //allStmts.add(blank);
            recursiveFindIterStmts((WsIteration)stmtsT.get(i));
        }
        else
            allStmts.add(stmtsT.get(i));
    }
}

/**
 * Recursively explores while loops to fine any nested statements
 * @param wsSelection initial while loop to search
 */
private static void recursiveFindIterStmts(WsIteration wsIteration)
{
    Vector<WsStatement> stmtsE = wsIteration.getWsIterBody();
    //WsStatement blank = new WsAssignment();
    for ( int i = 0; i < stmtsE.size(); i++)
    {
        if ( stmtsE.get(i) instanceof WsSelection)
        {
            //allStmts.add(blank);
            recursiveFindSelStmts((WsSelection)stmtsE.get(i));
```

```
        }
        else if ( stmtsE.get(i) instanceof WsIteration)
        {
           //allStmts.add(blank);
           recursiveFindIterStmts((WsIteration)stmtsE.get(i));
        }
        else
           allStmts.add(stmtsE.get(i));
     }
  }
```

XformMem1 Transform

```
/**
 * Source file: XformMem1.java
 * Purpose: apply the basic memento design pattern
 * <pre>
 * History:
 *    Original: 05-01-09 Gump created including methods:
 *              -XformAF1() default constructor
 *              -XformAF1(WsPackage) set up the ast
 *              -applicable(Object, OBject) check if can be applied
 *              -explain(Object) explain what will happen if can be applied
 *              -getDesc() description of transform
 *              -getName() name of xform
 *              -replay(WsPackage) not implemented
 *              -show() not implemented
 *              -toString() nothing really
 *              -undo(WsPackage, Object) not implemented
 *    Modified:
 * </pre>
 */

/**
    * Checks if memento can be applied.  To be applied, the params must be a
    * single WsClass containing some attributes.  In addition, all non-primitive
    * attribute types must have copy methods within the entire model and no
    * methods named "createMemento" or "setMemento" can exist in class.  Furthermore
    * the passed in "classNameMemento" cannot already exist in model.
    * @param tgt the ast to be modified
    * @param params the class to create memento suppot for
    * @return whether or not the pattern can be applied
    */
   public static boolean applicable(Object tgt, Object params)
   {
      //is it actually a class
      if ( !(params instanceof WsClass) )
      {
         System.out.println("Expected class to apply transform to!\n");
         return false;
      }

      WsClass cls = (WsClass)params;
      Vector<WsObject> attrs = cls.getAttributes();
      //are there any attributes
      if ( attrs == null )
      {
```

```
            System.out.println("Expected some attributes, none found!\n");
            return false;
        }


        WsPackage ast = (WsPackage)tgt;
        Vector<WsDeclaration> decls = ast.getWsDecls();
        //we begin by grabbing all the attributes that are class types
        //the other are other types like an array, so the current class should
        //have copy methods
        Vector<WsClass> classes = new Vector<WsClass>();
        Vector<WsMethod> meths = cls.getWsClassOperations();
        //short little intermission here, we first check that our class doesn't have
        //createMomento/setMomento methods!
        for ( int i = 0; i < meths.size(); i++)
        {
            if ( meths.get(i).getName().equals("createMemento") ||
meths.get(i).getName().equals("setMemento") )
            {
                System.out.println("create or set Momento method found, cannot create
another!");
                return false;
            }
        }
        //check if momento class exists
        WsClass testMem = (WsClass)XformUtils.getDeclByName((WsPackage)tgt,
cls.getName()+"Memento");
        if ( testMem != null )
        {
            System.out.println("A memento for this class was already found!");
            return false;
        }
        //find all class types
        for ( int i = 0; i < decls.size(); i++)
        {
            if ( decls.get(i) instanceof WsClass )
                classes.add((WsClass)decls.get(i));
        }
        //checking for copy methods where appropriate
        for ( int i = 0; i < attrs.size(); i++)
        {
            WsAttribute attr = (WsAttribute)attrs.get(i);
            //System.out.println(attr.getName() + " " + attr.getTypeName() + " ");
            if ( attr.getTypeName().equalsIgnoreCase("double") ||
attr.getTypeName().equalsIgnoreCase("int")
                || attr.getTypeName().equalsIgnoreCase("integer") ||
```

```
attr.getTypeName().equalsIgnoreCase("boolean")
                || attr.getTypeName().equalsIgnoreCase("bool"))
                ; //do nothing in this case, don't need a copy method
            //we have to search, is this a declared type (like array) or a class?
            else
            {
               String typeName = attr.getTypeName();
               boolean copyMethod = false;
               //search for the class with this name
               for ( int j = 0; j < classes.size(); j++)
               {
                  if ( typeName.equals(classes.get(j).getName()))
                  {
                     WsClass                                    matchC                                    =
(WsClass)XformUtils.getDeclByName((WsPackage)tgt, classes.get(j).getName());
                     //get the matching classes ops
                     meths = matchC.getWsClassOperations();
                     //see if this class has a way to copy the attributes
                     for ( int k = 0; k < meths.size(); k++)
                     {
                        if ( meths.get(k).getName().equals("copy"))
                        {
                           copyMethod = true;
                           break;
                        }
                     }
                  }
               }
               //didn't find a class that matched, might be a type declared
               //in this package, so the class itself should have a way to copy
               meths = cls.getWsClassOperations();
               for ( int k = 0; k < meths.size(); k++)
               {
                  //named copyAttributeName
                   if ( meths.get(k).getName().equals("copy" + attr.getName()))
                   {
                       copyMethod = true;
                       break;
                   }
               }
               if ( !copyMethod )
               {
                  System.out.println("Could not find a copy method for " + attr.getName());
                  return false;
               }
            }
```

```
    }
    return true;
  }

  /*****************************************************************
The Execute creates a constructor with the user specified attributes
as parameters.
It also assigns default values to the rest of the attributes.

Parameters:"params" is a vector. The first element of the params
                vector is the AST tree object of the parsed AWL file.
                (The "params" Vector is the same for both applicable
                and the execute)
                The rest of the elements of the params vector are Strings
                of user selected attributes. These attributes are to become
                parameters of the constructor being created.

NOTE: A more robust technique is needed to assign names to the parameters.
*****************************************************************/
public boolean createConstructor(WsClass targetclass, Vector<String> params) {

        WsFunction constructor = new WsFunction();
  constructor.setName(targetclass.getName());

        WsMethod m = new WsMethod(constructor);
        m.setWsPrivate(false);
        m.setWsClassMethod(true);

        targetclass.addWsClassOperation(m);

  Vector attr1 = targetclass.getAttributes();
        Vector attr = (Vector) attr1.clone();


        //Vector params1 = (Vector)params;

  //WsClasses.WsPackage myPac = (WsClasses.WsPackage) params1.get(0);
  Vector Decls = target.getWsDecls();

  WsClasses.WsAttribute a;
        WsClasses.WsAssignment assign;
        WsClasses.WsName name;
        WsClasses.WsExpression exp;
        WsClasses.WsDataType typeOfVar;
        WsClasses.WsDeclaration dec;
```

```
String var;

for(int l =0;l<params.size();l++)
{
        var = (String)params.get(l);
        for(int k=0; k<attr.size(); k++)
        {
                a = (WsClasses.WsAttribute) attr.get(k);

                if(var.equals(a.getName()))
                {
                        attr.remove(k);
                        //Need iterative increment to assure unique name.
                        WsClasses.WsParameter prm = new
WsClasses.WsParameter(a.getName()+ "1",a.getTypeName());
                        prm.setWsParameterIn(true);
                        constructor.addWsSubprogFormal(prm);

                        name = new WsClasses.WsIdentifierRef(a.getName());
                        exp = new WsIdentifierRef(prm.getName());

                        assign  = new WsClasses.WsAssignment(name, exp);

                        constructor.addWsSubprogBody(assign);

                }
                else if(k == attr.size()-1)
                        break;

        }

}//End of for


for(int i=0; i<attr.size(); i++)
{
        a = (WsClasses.WsAttribute) attr.get(i);

        for(int j=0; j<Decls.size(); j++)
        {
                dec = (WsClasses.WsDeclaration) Decls.get(j);

                if(a.getTypeName().equals(dec.getName()))
                {
                        if(dec instanceof WsIntegerType)
                        {
```

```
                                        exp = new WsClasses.WsLiteralInteger(0);

                                    }else if(dec instanceof WsRealType)
                                    {
                                        exp = new WsClasses.WsLiteralReal(0);

                                    }
                                    else
                                    {
                                        exp = new WsClasses.WsLiteralNull();
                                    }

                                    name = new WsClasses.WsIdentifierRef(a.getName());
                                    assign  = new WsClasses.WsAssignment(name, exp);

                                    constructor.addWsSubprogBody(assign);
                                }//End of if

            }//End of inner for-loop
        }//End of outer for-loop



      return true;

    }

    /**
     * This function is responsible for taking a specified class and a vector
     * of attributes in the class and creating get/set methods for them.
     * @param myclass the class to add get/set methods to
     * @param attrs the list of attributes we want get methods for
     */
    private void createGetsSets(WsClass myclass, Vector<WsAttribute> attrs)
    {
        //gets
        for ( int i = 0; i < attrs.size(); i++)
        {
            WsAttribute product = (attrs.get(i));
            WsFunction m = new WsFunction();
            WsMethod getProductFunc = new WsMethod(m);
            getProductFunc.setWsMethodSubprogram(m);
            m.setName("get" + product.getName());
            String identName = product.getTypeName();
            WsIdentifierRef returnIdent = new WsIdentifierRef(identName);
```

```
        m.setWsFunctReturnType(returnIdent);
        getProductFunc.setWsAbstract(false);
        getProductFunc.setWsClassMethod(false);
        getProductFunc.setWsPrivate(false);
        //the equivalent of AST's "return value"
        WsAssignment returnStmt = new WsAssignment();
        returnStmt.setWsAssignLHS(new WsIdentifierRef("get"+product.getName()));
        returnStmt.setWsAssignRHS(new WsIdentifierRef(product.getName()));
        m.addWsSubprogBody(returnStmt);
        myclass.addWsClassOperation(getProductFunc);
    }
    //sets
    for ( int i = 0; i < attrs.size(); i++)
    {
        //the formal parameter is the attribute
        WsParameter parmB = new WsParameter();
        parmB.setWsParameterType(attrs.get(i).getTypeName());
        parmB.setWsParameterIn(true);
        parmB.setWsParameterName(new WsIdentifier("the"+attrs.get(i).getName()));
        WsProcedure procB = new WsProcedure();
        String methName = new String("set" + attrs.get(i).getName());
        procB.setName(methName);
        procB.addWsSubprogFormal(parmB);
        WsMethod m = new WsMethod(procB);
        //the equivalent of AST's "return value"
        WsAssignment returnStmt = new WsAssignment();
        returnStmt.setWsAssignLHS(new WsIdentifierRef(attrs.get(i).getName()));
        returnStmt.setWsAssignRHS(new WsIdentifierRef(parmB.getName()));
        procB.addWsSubprogBody(returnStmt);
        myclass.addWsClassOperation(m);
    }
}


    /**
     * Applies the memento to the existing model passed into constructor.
     *
     * @param params
     * @return success or failure
     */
    @Override
    public boolean execute(Object params)
    {
        //don't bother trying to apply if not applicable
        if ( !applicable(target, params))
            return false;
```

```
//it has to be a class, checked by applicable
WsClass cls = (WsClass)params;

//stores where the copy method is
//0 - no copy needed 1 - in this class 2 - in an another object class
findCopies(cls);

//create the memento class
WsClass memento = new WsClass();
memento.setName(cls.getName()+"Memento");
Vector<WsAttribute> attrs = cls.getWsClassDataComponents();
Vector<WsAttribute> newAttrs = new Vector<WsAttribute>();
//Vector of strings needed to use other transform
Vector<String> stringAttrs = new Vector<String>();
//create new attributes in our class with same name/type
//but all must be private
for ( int i = 0; i < attrs.size(); i++)
{
   WsAttribute newAttr = new WsAttribute();
   newAttr.setName(attrs.get(i).getName());
   newAttr.setTypeName(attrs.get(i).getTypeName());
   newAttr.setWsPrivate(true);
   newAttrs.add(newAttr);
   stringAttrs.add(attrs.get(i).getName());
}
//give them to our class
memento.setWsClassDataComponents(newAttrs);

//make use of an existing transform function to create the constructor
//that takes all attributes
if ( !createConstructor(memento, stringAttrs) )
   return false;

//make use of an existing transform to create get methods for all
//attributes in the class
createGetsSets(memento, newAttrs);

target.addWsDecl(memento);

//create a set memento in the passed in class
WsProcedure setMem = new WsProcedure();
setMem.setName("setMemento");
//create argument list and add
WsParameter memParam = new WsParameter();
```

```
memParam.setWsParameterType(new WsIdentifierRef(memento.getName()));
memParam.setWsParameterOut(false);
memParam.setWsParameterName(new WsIdentifier("the"+memento.getName()));
setMem.addWsSubprogFormal(memParam);

//for each attribute, this is equal to classAttribute = memento.getAttribute()
for ( int i = 0; i < newAttrs.size(); i++)
{
   WsFunctionCall getCall = new WsFunctionCall();
   //theMemento.getAttr
   getCall.setName("the"+memento.getName()+".get"+newAttrs.get(i).getName());
   WsAssignment getAsgn = new WsAssignment();
   getAsgn.setWsAssignLHS(new WsIdentifierRef(newAttrs.get(i).getName()));
   getAsgn.setWsAssignRHS(getCall);
   setMem.addWsSubprogBody(getAsgn);
}
//actually make it a method
WsMethod m = new WsMethod(setMem);
m.setWsPrivate(false);
cls.addWsClassOperation(m);

//create the get memento in the passed in class to make mementos
WsFunction getMem = new WsFunction();
getMem.setName("getMemento");
getMem.setWsFunctReturnType(new WsIdentifierRef(memento.getName()));
//for each attribute, this stores it in a local variable by calling the
//appropriate copy method
//makeMem.setLHS(theMem);
Vector<WsVariable> formals = new Vector<WsVariable>();
//get all the locals we need
//this is the actual memento
 WsVariable localMem = new WsVariable();
 localMem.setName("localMem");
 localMem.setTypeName(memento.getName());
 formals.add(localMem);
 for ( int i = 0; i < newAttrs.size(); i++)
 {
    WsVariable local = new WsVariable();
    local.setName("local"+newAttrs.get(i).getName());
    local.setTypeName(newAttrs.get(i).getTypeName());
    formals.add(local);
 }


 ((WsSubprogram)getMem).setWsSubprogLocals(formals);
```

```
 //initialize all the variables with their values including the local Memento
 WsAssignment memAlloc = new WsAssignment();
 memAlloc.setWsAssignLHS(new WsIdentifierRef("localMem"));
 WsAllocator rhs = new WsAllocator();
 WsIdentifierRef tmp = new WsIdentifierRef(memento.getName());
 tmp.setWsIdentRefTo(memento.getWsDeclName());
 rhs.setWsAllocReturnType(tmp);
 memAlloc.setWsAssignRHS(rhs);
 getMem.addWsSubprogBody(memAlloc);
for ( int i = 0; i < newAttrs.size(); i++)
{
   WsAssignment set = new WsAssignment();
   //((WsSubprogram)getMem).setWsSubprogLocals(newAttrs);
   //set.setWsSubprogCallName("mem.set"+newAttrs.get(i).getName());
   set.setWsAssignLHS(new WsIdentifierRef(formals.get(i+1).getName()));

   //check where copy method is at
   //0 means primitive, no copy
   if ( copyLocation.get(i) == 0)
   {
     set.setWsAssignRHS(new WsIdentifierRef(newAttrs.get(i).getName()));
   }
   //1 means in the passed in class
   else if ( copyLocation.get(i) == 1)
   {
      WsFunctionCall copyMeth = new WsFunctionCall();
      copyMeth.setWsSubprogCallName("copy"+newAttrs.get(i).getName());
      set.setWsAssignRHS(copyMeth);
   }
   //2 means in another class
   else if ( copyLocation.get(i) == 2)
   {
      WsFunctionCall copyMeth = new WsFunctionCall();
      copyMeth.setWsSubprogCallName(newAttrs.get(i).getName()+".copy");
      set.setWsAssignRHS(copyMeth);
   }
   getMem.addWsSubprogBody(set);
}
for ( int i = 0; i < newAttrs.size(); i++)
{
   WsProcedureCall set = new WsProcedureCall();
   set.setWsSubprogCallName("localmem.set"+newAttrs.get(i).getName());
   Vector<WsIdentifierRef> args = new Vector<WsIdentifierRef>();
   //WsVariable tmp = new WsVariable();
   //tmp.setName(formals.get(i+1).getName());
   //tmp.setTypeName(formals.get(i+1).getTypeName());
```

```
        args.add(new WsIdentifierRef(formals.get(i+1).getName()));
        set.setWsSubprogCallArgs(args);
        getMem.addWsSubprogBody(set);
    }
    WsAssignment ret = new WsAssignment();
    ret.setWsAssignLHS(new WsIdentifierRef("getMemento"));
    ret.setWsAssignRHS(new WsIdentifierRef("localMemento"));
    getMem.addWsSubprogBody(ret);




    m = new WsMethod(getMem);

    cls.addWsClassOperation(m);

    return true;
}
```

XformObs1 Transform

```
/**
 * Source file: XformObs1.java
 * Purpose: apply the Obs1 transform to existing model.  Updates a model to
 * contain a shell of the observer pattern.
 * <pre>
 * History:
 *    Original: 07-18-09 Gump created including methods:
 *              -XformObs1() default constructor
 *              -XformObs1(WsPackage) set up the ast
 *              -applicable(Object, OBject) check if can be applied
 *              -explain(Object) explain what will happen if can be applied
 *              -getDesc() description of transform
 *              -getName() name of xform
 *              -replay(WsPackage) not implemented
 *              -show() not implemented
 *              -toString() nothing really
 *              -undo(WsPackage, Object) not implemented
 *    Modified:
 * </pre>
 */

/**
    * Checks if Obs1 can be applied.  To be applied, the model must not already contain
    * a class named Observable or Observer.  The params are unused.
    * @param tgt the ast to be modified
    * @param params unused
    * @return whether or not the pattern can be applied
    */
   public static boolean applicable(Object tgt, Object params)
   {
      if ( XformUtils.getDeclByName((WsPackage)tgt, "Observable") != null)
      {
         System.out.println("Observable class found, was pattern already applied?");
         return false;
      }
      if ( XformUtils.getDeclByName((WsPackage)tgt, "Observer") != null)
      {
         System.out.println("Observer class found, was pattern already applied?");
         return false;
      }

      return true;
   }
```

```
/**
 * Applies the Obs1 to the existing model passed into constructor.  In doing
 * so, creates a class named Observable that is abstract with attach(observer),
 * detach(observer), and notify() methods as well as array of Observers and
 * an abstract Observer class with an update() method is created.
 *
 * @param params
 * @return success or failure
 */
@Override
public boolean execute(Object params)
{
    WsClass observer = new WsClass();
    WsClass observable = new WsClass();

    //first we do observer class it is easy
    observer.setName("Observer");
    observer.setWsClassAbstract(true);
    //add the abstract update() method
    WsProcedure proc = new WsProcedure();
    proc.setName("update");
    WsMethod m = new WsMethod(proc);
    m.setWsAbstract(true);
    m.setWsPrivate(false);
    m.setWsClassMethod(false);
    observer.addWsClassOperation(m);


    //now for the interesting part of the observable class
    observable.setName("Observable");
    observable.setWsClassAbstract(true);
    //create the array to hold observers
    /*WsArrayType obs = new WsArrayType();
    obs.setIndexTypeName("int");
    obs.setElementTypeName("Observer");
    WsAttribute obsAttr = new WsAttribute("Variable", "observers", obs);
    obsAttr.setWsStatic(false);
    obsAttr.setWsPrivate(true);
    observable.addWsClassDataComponent(obsAttr);*/

    //attach(Observer)
    proc = new WsProcedure();
    proc.setName("attach");
    WsParameter input = new WsParameter();
    input.setWsParameterIn(true);
```

```
    input.setWsParameterName("o");
    input.setWsParameterType("Observer");
    proc.addWsSubprogFormal(input);
    m = new WsMethod(proc);
    m.setWsAbstract(true);
    m.setWsPrivate(false);
    m.setWsClassMethod(false);
    observable.addWsClassOperation(m);

    //detach(Observer)
    proc = new WsProcedure();
    proc.setName("detach");
    input = new WsParameter();
    input.setWsParameterIn(true);
    input.setWsParameterName("o");
    input.setWsParameterType("Observer");
    proc.addWsSubprogFormal(input);
    m = new WsMethod(proc);
    m.setWsAbstract(true);
    m.setWsPrivate(false);
    m.setWsClassMethod(false);
    observable.addWsClassOperation(m);

    //notify()
    proc = new WsProcedure();
    proc.setName("notify");
    m = new WsMethod(proc);
    m.setWsAbstract(true);
    m.setWsPrivate(false);
    m.setWsClassMethod(false);
    observable.addWsClassOperation(m);

    target.addWsDecl(observer);
    target.addWsDecl(observable);
    return true;
}
```

XformObs2 Transform

```
/**
 * Source file: XformObs2.java
 * Purpose: apply the Obs2 transform to existing model.  This transform takes a
 * specified class and makes its parent class the Observer class and ads the update method
 * the class requires.
 * <pre>
 * History:
 *    Original: 07-28-09 Gump created including methods:
 *            -XformObs2() default constructor
 *            -XformObs2(WsPackage) set up the ast
 *            -applicable(Object, OBject) check if can be applied
 *            -explain(Object) explain what will happen if can be applied
 *            -getDesc() description of transform
 *            -getName() name of xform
 *            -replay(WsPackage) not implemented
 *            -show() not implemented
 *            -toString() nothing really
 *            -undo(WsPackage, Object) not implemented
 *    Modified:
 * </pre>
 */

/**
    * Checks if Obs2 can be applied.  To be applied, the model must already contain
    * a class Observer with abstract update method.  The param should be classes
    * to add the parent to, as such they must not have any super class yet or a method
    * called update.
    * @param tgt the ast to be modified
    * @param params the classes to be checked and modified
    * @return whether or not the pattern can be applied
    */
   public static boolean applicable(Object tgt, Object params)
   {
      //see if there is an existing observer class with update method
      try
      {
         WsClass      obs      =      (WsClass)XformUtils.getDeclByName((WsPackage)tgt,
"Observer");
         if ( obs == null)
         {
            System.out.println("Observer class expected, but not found!");
            return false;
         }
         Vector<WsMethod> oMeths = obs.getWsClassOperations();
```

```
            boolean update = false;
            for ( int i = 0; i < oMeths.size(); i++)
            {
               if (oMeths.get(i).getName().equals("update"))
                  update = true;
            }
            if ( !update )
            {
               System.out.println("Update method not found, required.");
               return false;
            }
         }
         catch (Exception e)
         {
            System.out.println("Expected class observer, not found.");
            return false;
         }

         //see if there is an observable class with attach/detach/update methods
         //see if there is an existing observer class with update method
         try
         {
            WsClass obs = (WsClass)XformUtils.getDeclByName((WsPackage)tgt,
"Observable");
            if ( obs == null)
            {
               System.out.println("Observable class expected, but not found!");
               return false;
            }
            Vector<WsMethod> oMeths = obs.getWsClassOperations();
            boolean update = false, attach = false, detach = false;
            for ( int i = 0; i < oMeths.size(); i++)
            {
               if (oMeths.get(i).getName().equals("notify"))
                  update = true;
               if (oMeths.get(i).getName().equals("detach"))
                  detach = true;
               if (oMeths.get(i).getName().equals("attach"))
                  attach = true;
            }
            if ( !update || !attach || !detach )
            {
               System.out.println("Required methods not found, cannot apply.");
               return false;
            }
         }
```

```
        catch (Exception e)
        {
           System.out.println("Expected class observable, not found.");
           return false;
        }

        //see if there are classes to update
        try
        {
           Vector<String> classes = (Vector<String>)params;
           if ( classes.size() < 1 )
           {
              System.out.println("Expected at least 1 class to update!");
              return false;
           }
           for ( int i = 0; i < classes.size(); i++)
           {
              WsClass cls = (WsClass)XformUtils.getDeclByName((WsPackage)tgt,
classes.get(i));
              //they should not have a parent already
              if ( cls == null )
              {
                 System.out.println("Expected existing class with no super class.");
                 return false;
              }
              else if ( cls.getSuperclassName() != null &&
!cls.getSuperclassName().trim().equals("") )
              {
                  System.out.println("Expected existing class with no super class.");
                  return false;
              }
              else if ( cls.getName().contains("Observer") ||
cls.getName().equals("Observable"))
              {
                 System.out.println("Observer and Observable class should not subclass
Observer!");
                 return false;
              }
              else //they should not have a method called update yet
              {
                 Vector<WsMethod> meths = cls.getWsClassOperations();
                 for ( int j = 0; j < meths.size(); j++)
                 {
                    if ( meths.get(j).getName().trim().equals("update"))
                    {
                       if ( meths.get(j).getFormals() != null &&
```

```
meths.get(j).getFormals().size() == 1)
                    {
                    System.out.println("update method already found!");
                    return false;
                    }
                }
            }
        }
    }
    catch (Exception e)
    {
        System.out.println("Expected a list of classes to apply update to." + e);
        return false;
    }

    return true;
}


/**
 * Applies the Obs2 to the existing model passed into constructor.  In doing
 * so, takes each of the specified classes and makes the parent class the Observer
 * class and adds the require update method to the class.     *
 * @param params
 * @return success or failure
 */
@Override
public boolean execute(Object params)
{
    if ( !applicable(target, params))
        return false;

    Vector<String> sCls = (Vector<String>)params;
    Vector<WsClass> classes = new Vector<WsClass>();
    //convert strings to classes
    for ( int i = 0; i < sCls.size(); i++)
    {
        classes.add((WsClass)XformUtils.getDeclByName(target, sCls.get(i)));
    }

    //add super class of observer
    //and add update method
    for ( int i = 0; i < classes.size(); i++)
    {
        classes.get(i).setSuperclassName("Observer");
```

```
        WsProcedure update = new WsProcedure();
        update.setName("update");
        WsMethod m = new WsMethod(update);
        m.setWsAbstract(false);
        m.setWsClassMethod(false);
        m.setWsPrivate(false);
        classes.get(i).addWsClassOperation(m);
    }
    return true;
}
```

XformVis1 Transform

```
/**
 * Source file: XformVis1.java
 * Purpose: apply the vis1 transform.  This transform takes an existing
 * set of classes and a String name and creates a abstract superclass with the given
 * name that the specified classes inherit from.
 * <pre>
 * History:
 *    Original: 07-11-09 Gump created including methods:
 *            -XformVis1() default constructor
 *            -XformVis1(WsPackage) set up the ast
 *            -applicable(Object, OBject) check if can be applied
 *            -explain(Object) explain what will happen if can be applied
 *            -getDesc() description of transform
 *            -getName() name of xform
 *            -replay(WsPackage) not implemented
 *            -show() not implemented
 *            -toString() nothing really
 *            -undo(WsPackage, Object) not implemented
 *    Modified:
 * </pre>
 */

/**
    * Checks if vis1 can be applied.  Cannot be applied if classes already
    * inherit from something or the name specified is in use.
    * @param tgt the ast to be modified
    * @param params a vector with a list of classes followed by a string name
    * for the new super class
    * @return whether or not the pattern can be applied
    */
   public static boolean applicable(Object tgt, Object params)
   {
      //is this a list of arguments
      if ( !(params instanceof Vector) )
      {
         System.out.println("Expected list of classes.");
         return false;
      }

      //convert to a vector
      Vector args = (Vector)params;

      //need at least one concrete class and super class name
      if ( args.size() < 2)
```

```
        {
            System.out.println("Expected at least one class and super class name.");
            return false;
        }

        //get all the subclasses into a seperate vector
        Vector<WsClass> subClasses = new Vector<WsClass>();
        try
        {
            for ( int i = 0; i < args.size() - 1; i++ )
                subClasses.add((WsClass)args.get(i));
        }
        catch ( Exception e ) //in case they were not all classes
        {
            System.out.println("Expected classes in list; did not find.");
            return false;
        }

        for ( int i = 0; i < subClasses.size(); i++)
        {
            if ( XformUtils.getDeclByName((WsPackage)tgt, subClasses.get(i).getName())
== null )
            {
                System.out.println("The specified class wasn't in the model... how did that
happen!?");
                return false;
            }
        }
        //get the name of the super class
        String superClassName;
        try
        {
            superClassName = (String)args.get(args.size()-1);
        }
        catch ( Exception e)
        {
            System.out.println("Expected name of new class as last arg!");
            return false;
        }

        //is the name unique?
        if ( XformUtils.getDeclByName((WsPackage)tgt, superClassName) != null )
        {
            System.out.println("Expected unique name, but " + superClassName
                    + " was in use.");
            return false;
```

```
        }

        //they don't already inherit do they
        for ( int i = 0; i < subClasses.size(); i++ )
        {
           if ( subClasses.get(i).getSuperclassName() != null &&
!subClasses.get(i).getSuperclassName().equals(""))
           {
              System.out.println("Subclass " + subClasses.get(i).getName() + " already " +
                    "inherits from a class, cannot implement multiple inheritance.");
              return false;
           }
        }
        return true;
     }


     /**
      * Applies the vis1 to the existing model passed into constructor.  This will
      * create a new abstract super class with the name specified as the last
      * argument in the list that is params.  All the prior arguments should be classes
      * and will have their parent class set to the new class     *
      * @param params a vector of input with all classes and the last element a
      * string.
      * @return success or failure
      */
     @Override
     public boolean execute(Object params)
     {
        //don't bother trying to apply if not applicable
        if ( !applicable(target, params))
           return false;

         //convert to a vector
        Vector args = (Vector)params;

        //get all the subclasses into a seperate vector
        Vector<WsClass> subClasses = new Vector<WsClass>();
        for ( int i = 0; i < args.size() - 1; i++ )
              subClasses.add((WsClass)args.get(i));

        //get the name of the super class
        String superClassName = (String)args.get(args.size()-1);

        WsClass superC = new WsClass();
        superC.setName(superClassName);
```

```
    superC.setWsClassAbstract(true);
    superC.setWsDeclName(new WsIdentifier(superClassName));

    //tell the babies they have a new parent
    for ( int i = 0; i < subClasses.size(); i++ )
    {
       subClasses.get(i).setParent(superC);
       subClasses.get(i).setSuperclassName(superClassName);
    }

    target.addWsDecl(superC);
    return true;
}
```

XformVis2 Transform

```
/**
 * Source file: XformVis2.java
 * Purpose: apply the Vis2 transform to existing model.  This transform requires
 * that the Vis3 transform should have been applied (the visitor pattern itself)
 * and will add a specified attribute of a given type to the visit methods
 * to allow extra information to be passed.  Also updates the accept method
 * to call with the new argument, initially assumed null.
 * <pre>
 * History:
 *    Original: 07-11-09 Gump created including methods:
 *            -XformVis2() default constructor
 *            -XformVis2(WsPackage) set up the ast
 *            -applicable(Object, OBject) check if can be applied
 *            -explain(Object) explain what will happen if can be applied
 *            -getDesc() description of transform
 *            -getName() name of xform
 *            -replay(WsPackage) not implemented
 *            -show() not implemented
 *            -toString() nothing really
 *            -undo(WsPackage, Object) not implemented
 *    Modified:
 * </pre>
 */

/**
    * Checks if vis2 can be applied.  To be applied, the two classes specified
    * must actually exist in the model.  Also need a new attribute name and type!
    * the type must exist in the model if it is not a primitive
    * @param tgt the ast to be modified
    * @param params vector with four strings, first class name of visitor class,
    * second is name of super class for inheritance hierarchy, third is new attribute
    * type, fourth is new attribute name
    * @return whether or not the pattern can be applied
    */
   public static boolean applicable(Object tgt, Object params)
   {
      if ( !(params instanceof Vector) )
      {
         System.out.println("Expected vector with four strings!");
         return false;
      }

      String visName = "", acceptName = "", type = "", name = "";
      //make sure only two args that are both strings
```

```
      try
      {
         Vector<String> args = (Vector<String>)params;
         if (args.size() != 4)
            throw new Exception();
         visName = args.get(0);
         acceptName = args.get(1);
         type = args.get(2);
         name = args.get(3);
         if ( visName.equals("") || type.equals("") || name.equals("") ||
acceptName.equals(""))
            throw new Exception();
      }
      catch (Exception e)
      {
         System.out.println("Expected four arguments as strings for class names of" +
               " the visitor super class and class hiearchy super class.  Also need" +
               "new attribute and type of it.");
         return false;
      }

      //make sure the two classes actually exist in the model
      try
      {
         WsClass visC = (WsClass)XformUtils.getDeclByName((WsPackage)tgt,
visName);
         WsClass acceptC = (WsClass)XformUtils.getDeclByName((WsPackage)tgt,
acceptName);
         WsDeclaration objectD = XformUtils.getDeclByName((WsPackage)tgt, type);
         if ( visC == null || acceptC == null || objectD == null )
            throw new Exception();
      }
      catch (Exception e)
      {
         System.out.println("Excepted the class names/object type specified to exist in
model!");
         return false;
      }

      return true;
   }


   /**
    * Applies the vis2 to the existing model passed into constructor.
    *
```

```
    * @param params
    * @return success or failure
    */
   @Override
   public boolean execute(Object params)
   {
      //don't bother trying to apply if not applicable
      if ( !applicable(target, params))
         return false;

      //get the super class names
      Vector<String> classes = (Vector<String>)params;
      String visN = classes.get(0);
      String superN = classes.get(1);
      String type = classes.get(2);
      String name = classes.get(3);

      Vector<String>  subVisitors  =  WsTools.ToolUtils.returnChildrenNames(target,
visN);
      Vector<String>  subSupers  =  WsTools.ToolUtils.returnChildrenNames(target,
superN);

      //this will go through visitor classes and add to them the new attribute
      for ( int i = 0; i < subVisitors.size(); i++ )
      {
         WsClass cls = (WsClass)XformUtils.getDeclByName(target, subVisitors.get(i));
         for ( int j = 0; j < subSupers.size(); j++ )
         {
            //try to find the visit method in the class, should be called VisitCLASSNAME
            WsClass  otherCls  =  (WsClass)XformUtils.getDeclByName(target,
subSupers.get(j));
            WsMethod meth = cls.getWsClassOperation("Visit"+otherCls.getName());
            WsParameter arg = new WsParameter();
            arg.setWsName(new WsIdentifier(name));
            arg.setWsParameterType(type);
            arg.setWsParameterIn(true);
            arg.setWsParameterOut(false);
            WsSubprogram subMeth = meth.getWsMethodSubprogram();
            subMeth.addWsSubprogFormal(arg);
         }
      }
      //we also have to update the abstract class too
      WsClass cls = (WsClass)XformUtils.getDeclByName(target, visN);
      for ( int j = 0; j < subSupers.size(); j++ )
      {
         //try to find the visit method in the class, should be called VisitCLASSNAME
```

```
        WsClass otherCls = (WsClass)XformUtils.getDeclByName(target,
subSupers.get(j));
        WsMethod meth = cls.getWsClassOperation("Visit"+otherCls.getName());
        WsParameter arg = new WsParameter();
        arg.setWsName(new WsIdentifier(name));
        arg.setWsParameterType(type);
        WsSubprogram subMeth = meth.getWsMethodSubprogram();
        subMeth.addWsSubprogFormal(arg);
      }

      //now the tricky part, have to update the calls in the accept methods
      //to now do the equivalent of vistor.visit(this, null) where null could be an Object
      for ( int i = 0 ; i < subSupers.size(); i++)
      {
        cls = (WsClass)XformUtils.getDeclByName(target, subSupers.get(i));
        Vector<WsMethod> meths = cls.getWsClassOperations();
        for ( int j = 0; j < meths.size(); j++)
        {
          //is it an Accept method
          if ( meths.get(j).getName().equals("Accept"))
          {
            Vector<WsParameter> args = meths.get(j).getFormals();
            //does it have arg that is our type
            if ( args != null && args.get(0).getType().equals(visN))
            {
              //now we have to find that old statement
              Vector<WsStatement>                    stmts                    =
meths.get(j).getWsMethodSubprogram().getWsSubprogBody();
              //is it a procedure call?
              for ( int k = 0; k < stmts.size(); k++)
              {
                if ( stmts.get(k) instanceof WsProcedureCall)
                {
                  WsProcedureCall call = (WsProcedureCall)stmts.get(k);
                  if                                                                (
call.getWsSubprogCallName().getName().equals("visitor.Visit"+subSupers.get(i)))
                  {
                    //call.addWsSubprogCallArg(new WsIdentifierRef("null"));
                    call.addWsSubprogCallArg(new WsLiteralNull());
                  }
                }
              }
              WsParameter param = new WsParameter();
              param.setWsName(new WsIdentifier(name));
              param.setWsParameterIn(true);
              param.setWsParameterOut(false);
```

```
                param.setWsParameterType(type);
                WsSubprogram prog = meths.get(j).getWsMethodSubprogram();
                prog.addWsSubprogFormal(param);
            }
          }
        }
      }

      //we have to do the same thing for the abstract parent class... man this
      //really freaking sucks a whole lot
      cls = (WsClass)XformUtils.getDeclByName(target, superN);
      Vector<WsMethod> meths = cls.getWsClassOperations();
      for ( int i = 0; i < meths.size(); i++ )
      {
        //looking for accept method
        if ( meths.get(i).getName().equals("Accept"))
        {
          //looking for only one argument
          if ( meths.get(i).getFormals() != null)// && meths.get(i).getFormals().size() ==
1)
          {
            //of same of the specified visitor class type
            Vector<WsParameter> paramList = meths.get(i).getFormals();
            if ( paramList.get(0).getType().equals(visN))
            {
              WsParameter param = new WsParameter();
              param.setWsName(new WsIdentifier(name));
              param.setWsParameterIn(true);
              param.setWsParameterOut(false);
              param.setWsParameterType(type);
              WsSubprogram prog = meths.get(i).getWsMethodSubprogram();
              prog.addWsSubprogFormal(param);
            }
          }
        }
      }
      return true;
    }
```

XformVis3 Transform

```
/**
 * Source file: XformVis3.java
 * Purpose: apply the basic visitor design pattern
 * <pre>
 * History:
 *    Original: 07-11-09 Gump created including methods:
 *              -XformVis3() default constructor
 *              -XformVis3(WsPackage) set up the ast
 *              -applicable(Object, OBject) check if can be applied
 *              -explain(Object) explain what will happen if can be applied
 *              -getDesc() description of transform
 *              -getName() name of xform
 *              -replay(WsPackage) not implemented
 *              -show() not implemented
 *              -toString() nothing really
 *              -undo(WsPackage, Object) not implemented
 *    Modified:
 * </pre>
 */

/**
    * Checks if vis3 can be applied.  Can only be applied if the specifed
    * arguments are a list of classes with the same super class and the last
    * element in the params is a string name for a class that does not exist
    * in model when concatenated with word "Visitor"
    * @param tgt the ast to be modified
    * @param params a vector of class names followed by a string
    * @return whether or not the pattern can be applied
    */
   public static boolean applicable(Object tgt, Object params)
   {

      //is this a list of arguments
      if ( !(params instanceof Vector) )
      {
         System.out.println("Expected list of classes.");
         return false;
      }

      //convert to a vector
      Vector args = (Vector)params;

      //need at least one concrete class and new class name
      if ( args.size() < 2)
```

```
        {
            System.out.println("Expected at least one class and visitor class name.");
            return false;
        }

        //get all the classes into a seperate vector, the 1st one is the super class
        Vector<WsClass> classes = new Vector<WsClass>();
        try
        {
            for ( int i = 0; i < args.size() - 1; i++ )
                classes.add((WsClass)args.get(i));
        }
        catch ( Exception e ) //in case they were not all classes
        {
            System.out.println("Expected classes in list; did not find.");
            return false;
        }

        //get the name of the super class
        String visitorClassName;
        try
        {
            visitorClassName = (String)args.get(args.size()-1);
        }
        catch ( Exception e)
        {
            System.out.println("Expected name of new class as last arg!");
            return false;
        }

        //is the name+"Visitor" unique?
        if ( XformUtils.getDeclByName((WsPackage)tgt, visitorClassName + "Visitor") !=
null )
        {
            System.out.println("Expected unique name, but " + visitorClassName
                    + " was in use.");
            return false;
        }

        //the subclasses must inherit from the specified super class
        String superClass = classes.get(0).getSuperclassName();
        if ( superClass == null || superClass.equals(""))
        {
            System.out.println("Common super class not found!");
            return false;
        }
```

```
      //make sure all classes inherit from the same parent
      for ( int i = 0; i < classes.size(); i++ )
      {
         if ( classes.get(i).getSuperclassName() == null ||
!classes.get(i).getSuperclassName().equals(superClass))
         {
            System.out.println("Subclass " + classes.get(i).getName() + " does not " +
                  " inherit from " + superClass + ".");
            return false;
         }
      }
      //make sure the abstract super class doesn't have an accept method already
      WsClass superC;
      try
      {
         superC = (WsClass)XformUtils.getDeclByName((WsPackage)tgt, superClass);
         if ( superC == null )
         {
            System.out.println("Super class " + superClass + " didn't exist!");
            return false;
         }
      }
      catch (Exception e)
      {
         System.out.println("Super class " + superClass + " didn't exist!");
         return false;
      }

      //make sure the super class does not already have an accept(classNamevisitor)
method
      WsMethod accept = superC.getWsClassOperation("Accept");
      if ( accept != null )
      {
            Vector<WsParameter> vars = accept.getFormals();
            if ( vars != null && vars.size() == 1 &&
vars.get(0).getType().equals(visitorClassName+"Visitor"))
            {
               System.out.println("The accept method already existed in super class, cannot
create it.");
               return false;
            }
      }
      //make sure the classes do not already have an accept(classNamevisitor) method
      for ( int i = 0; i < classes.size(); i++ )
      {
```

```
            accept = classes.get(i).getWsClassOperation("Accept");
            if ( accept != null )
            {
               Vector<WsParameter> vars = accept.getFormals();
               if ( vars != null && vars.size() == 1 &&
vars.get(0).getType().equals(visitorClassName+"Visitor"))
               {
                  System.out.println("The accept method already existed, cannot create it.");
                  return false;
               }
            }
         }
         return true;
      }


      /**
       * Applies the vis3 to the existing model passed into constructor.  This applies
       * the visitor pattern to an existing set of classes with a common super class.
       * This is accomplished by creating an abstract visitor class with visit methods
       * for each of the specified subclasses (whose name is derived from the last element
       * in the params list + the word "Visitor".  Also, all the specified classes
       * and their superclass receive accept methods that take as an argument the visitor
       * class.
       *
       * @param params
       * @return success or failure
       */
      @Override
      public boolean execute(Object params)
      {
         //don't bother trying to apply if not applicable
         if ( !applicable(target, params))
            return false;

        Vector args = (Vector)params;

         //get all the classes into a seperate vector, the 1st one is the super class
         Vector<WsClass> classes = new Vector<WsClass>();
         for ( int i = 0; i < args.size() - 1; i++ )
               classes.add((WsClass)args.get(i));

         //get the name of the super class
         String visitorClassName;
         visitorClassName = (String)args.get(args.size()-1);
         //create the class
```

```
WsClass visitor = new WsClass();
visitor.setName(visitorClassName + "Visitor");
visitor.setWsDeclName(new WsIdentifier(visitorClassName+"Visitor"));
visitor.setWsClassAbstract(true);
//make all the visit methods
Vector<WsMethod> visits = new Vector<WsMethod>();
for ( int i = 0; i < classes.size(); i++ )
{
   //create visit method
   WsProcedure visitM = new WsProcedure();
   visitM.setName("Visit" + classes.get(i).getName());
   //create an argument that is one of the concrete subclasses
   WsParameter visParam = new WsParameter();
   visParam.setWsParameterType(new WsIdentifierRef(classes.get(i).getName()));
   visParam.setWsParameterOut(false);
   visParam.setWsParameterIn(true);
   visParam.setWsParameterName(new WsIdentifier("node"));
   visitM.addWsSubprogFormal(visParam);
   WsMethod m = new WsMethod(visitM);
   m.setWsAbstract(true);
   visits.add(m);
}
//add all the functions to the class
visitor.setWsClassOperations(visits);
//add the class to the model
target.addWsDecl(visitor);

//update the specified classes to have their accept methods
for ( int i = 0; i < classes.size(); i++ )
{
   WsProcedure acceptM = new WsProcedure();
   //just called accept
   acceptM.setName("Accept");
   //create an argument that is the abstract visitor super class
   WsParameter acceptParam = new WsParameter();
   acceptParam.setWsParameterType(new WsIdentifierRef(visitor.getName()));
   acceptParam.setWsParameterOut(false);
   acceptParam.setWsParameterIn(true);
   acceptParam.setWsParameterName(new WsIdentifier("visitor"));
   acceptM.addWsSubprogFormal(acceptParam);

   //create equivalent call of visitor.acceptClassName(this)
   WsProcedureCall procCall = new WsProcedureCall();
   procCall.setName("visitor.Visit"+classes.get(i).getName());
   //have to make a list even though only one!
   Vector<WsExpression> param = new Vector<WsExpression>();
```

```
            //WsParameter thisP = new WsParameter();
            //thisP.setWsName(new WsIdentifier("this"));
            //thisP.setWsParameterIn(false);
            //thisP.setWsParameterType("");
            //param.add(new WsIdentifierRef("this"));
            param.add(new WsThis());
            procCall.setWsSubprogCallArgs(param);
            acceptM.addWsSubprogBody(procCall);
            WsMethod m = new WsMethod(acceptM);
            m.setWsAbstract(false);
            classes.get(i).addWsClassOperation(m);
        }
        //update the super class to have it too!
        WsClass superC = (WsClass)XformUtils.getDeclByName(target,
classes.get(0).getSuperclassName());
        WsProcedure acceptM = new WsProcedure();
        //just called accept
        acceptM.setName("Accept");
        //create an argument that is the abstract visitor super class
        WsParameter acceptParam = new WsParameter();
        acceptParam.setWsParameterType(new WsIdentifierRef(visitor.getName()));
        acceptParam.setWsParameterOut(false);
        acceptParam.setWsParameterIn(true);
        acceptParam.setWsParameterName(new WsIdentifier("visitor"));
        acceptM.addWsSubprogFormal(acceptParam);
        WsMethod m = new WsMethod(acceptM);
        m.setWsAbstract(true);
        superC.addWsClassOperation(m);


      return true;
    }
```

Tool Utilities Update

```
/*****************************************************************
* CLASS: ToolUtils
* DESCRIPTION: contains AwSome utility functions.
* Contains:
*  static boolean canDoAssocObject(WsAssocObject assoc, WsPackage myAST)
*  static boolean canDoAssocToObject(WsAssociation assoc, WsPackage myAST)
*  static boolean canDoOneWay(WsAssociation assoc, WsClass fromClass, WsPackage
myAST)
*  static Vector getClassNames(WsAssociation assoc)
*  static Vector getObjectClassNames(WsAssocObject assoc)
*  static WsAssociation getWsAssociation(WsPackage myAST, String assocName)
*  static WsAssocObject getWsAssocObject(WsPackage myAST, String assocName)
*  static WsExpression parseAssociation(String assocString)
*  static WsExpression parseExpression(String expString)
*  static WsAssociation promptAndGetAssoc(WsPackage ast, String prompt)
*  static WsAssociation promptAndGetAssocObject(WsPackage ast, String prompt)
*  static WsClass promptAndGetClass(WsPackage ast, String prompt)
*  static WsExpression promptAndGetExpression(String prompt)
*  static WsMethod promptAndGetMethod(WsClass aclass, String prompt)
*  static void reLink(WsPackage ast) - DOESN'T WORK AS INTENDED.
*  static Vector returnRoleNames(WsPackage myAST)
*  static WsClass returnWsClass(WsPackage myAST, String className)
*  static String toAWLstring(WsObject anObj)
* PRATAP ADDITIONS:
*  static Vector returnExpressions(WsExpression exp1)
*  static Vector returnClassNames(WsPackage MyAST)
*  static WsExpression returnChangedExp(WsExpression exp,String name)
* GUMP ADDITIONS:
*  static Vector returnChildrenNames(WsPackage MyAST, String parent)
* IP:
*  static void main(String[] args)
* Private:
*  NONE
* Inherited:
*  NONE
* History:
*  11/08/2004 - Hartrum - Original.
*  01/07/2005 - Hartrum - Changed ToolLoader to toolbox.ToolLoader in
*               main.  NEED A LOADER IN WsTools.
*  02/08/2005 - Hartrum - added promptAndGetMethod().
*  02/10/2005 - Hartrum - added toAWLstring().
*  02/15/2005 - Hartrum - added promptAndGetExpression().
*  02/17/2005 - Hartrum - added parseExpression().
*  05/14/2005 - Sarvepalli - added returnVarNames().
```

```
*  05/15/2005 - Sarvepalli - added returnTickExp().
*  09/19/2005 - Hartrum - cleaned up Sarvepalli's code
*  03/01/2006 - Hartrum - more clean up of Sarvepalli's code
*  03/08/2006 - Hartrum - changed name returnTickExp() to makeTickExp().
*  03/08/2006 - Hartrum - moved makeTickExp() to WsTransforms.XformUtils.
*  03/13/2006 - Hartrum - added Pratap's returnExpressions() & returnChangedExp().
*  03/13/2006 - Hartrum - added Pratap's returnClassNames().
*  03/08/2006 - Hartrum - moved returnVarNames() to WsTransforms.XformUtils.
*  09/19/2007 - Hartrum - cleaned up header documentation.
*  11/30/2007 - Hartrum - added reLink().
*  08/27/2008 - Hartrum - added Swamy's getClassNames(),
*             getObjectClassNames(), promptAndGetAssocObject().
*  09/05/2008 - Hartrum - added Swamy's canDo(), returnRoleNames(),
*             import WsUtilities.
*  09/09/2008 - Hartrum - renamed canDo() to canDoOneWay(),
*  09/10/2008 - Hartrum - added Swamy's canDoAssocToObject(), canDoAssocObject().
*  09/11/2008 - Hartrum - added Swamy's returnWsClass(), getWsAssocObject(), and
*             getWsAssociation().
*  10/07/2008 - Hartrum - clarified println() statements.
*  11/24/2008 - Hartrum - added parseAssociation().
*  05/04/2009 - Gump - added returnChildrenNames
**********************************************************************/


 /**
  * Search the passed ast for all children of the class with specified name.
  * @param MyAST the entire model to be searched
  * @param parent the name of the super class we are looking for children that
  * extend
  * @return a vector of strings that represent class names of classes that
  * had the passed parent
  */
 public static Vector returnChildrenNames(WsPackage MyAST, String parent)
 {
    Vector<String> children = new Vector<String>();
    //get all classes
    Vector<String> allClasses = XformUtils.returnClassNames(MyAST);
    for ( int i = 0; i < allClasses.size(); i++)
    {
       WsClass theClass = (WsClass)XformUtils.getDeclByName(MyAST,
allClasses.get(i));
       if ( theClass.getSuperclassName() != null &&
theClass.getSuperclassName().equals(parent) )
          children.add(theClass.getName());
    }
    return children;
```

```
}
```

Transform Utilities Update

```
/*****************************************************************
* CLASS: XformUtils
* DESCRIPTION: contains Xform utility functions.
* Contains:
*  static boolean       doesDeclNameExist(WsPackage top, String name)
*  static WsPackage     findPackage(WsPackage top, String name)
*  static WsDeclaration getDeclByName(WsPackage top, String name)
*  static boolean       isDeclInUse(WsPackage top, String name)
*  static WsExpression  makeTickExp(WsExpression exp)
*  static WsExpression  pathToVar(WsExpression path)
*  static Vector returnAggNames(WsPackage top)
*  static Vector returnAssocNames(WsPackage top)
*  static Vector returnAssocObjNames(WsPackage top)
*  static Vector returnClassNames(WsPackage top)
*  static Vector returnMethodNames(WsClass cls)
*  static Vector returnVarNames(WsExpression exp, boolean tick)
*  static WsExpression  varToPath(WsClass ctx, WsExpression tgt)
*  GUMP ADDITIONS:
*  static String makeUniqueClassName(WsPackage myAST, String base)
* Private:
*  NONE
* Inherited:
*  NONE
* History:
*  12/06/2004 - Hartrum - Original w/ findPackage().
*  12/07/2004 - Hartrum - Added getDeclByName() & doesDeclNameExist().
*  12/08/2004 - Hartrum - Added isDeclInUse().
*  01/21/2005 - Hartrum - Added import toolbox.*; Need to fix this
*  03/08/2006 - Hartrum - Added makeTickExp().
*  03/20/2006 - Hartrum - Added returnVarNames().
*  04/17/2006 - Hartrum - Added returnClassNames().
*  04/18/2006 - Hartrum - Added returnMethodNames().
*  11/05/2007 - Sneha   - Added substitute().
*  09/04/2008 - Hartrum - Added Swamy's returnAssocNames(),
*               returnAssocObjNames(), and returnAggNames().
*  11/20/2008 - Hartrum - Added pathToVar().
*  11/26/2008 - Hartrum - Added varToPath().
*  05/04/2009 - Gump - Added makeUniqueClassName()
*****************************************************************/

/**
   * Takes the specified base and appends random characters to the end such
   * that the resultant name will form a unique class name in the passed AST.
   * @param myAST the entire model to be searched for class names
```

```
 * @param base the base part of the class name to be added on to
 * @return original base contents + random characters
 */
public static String makeUniqueClassName(WsPackage myAST, String base)
{
   String tmp = "";
   Random rand = new Random();
   boolean unique = false;
   while (!unique)
   {
      unique = true;
      //add 1-5 new characters
      int charsToAdd = rand.nextInt(5)+1;
      tmp = base;
      for ( int i = 0; i < charsToAdd; i++)
         tmp += (char)(rand.nextInt(26)+65); //get an ASCII charactet A-Z

      Vector<String> existingClasses = XformUtils.returnClassNames(myAST);
      for ( int i = 0; i < existingClasses.size(); i++)
          if ( existingClasses.get(i).equals(tmp))
          {
             unique = false;
             break;
          }
   }
   return tmp;
}
```

BIBLIOGRAPHY

[1] Reutter, John, "Maintenance is a Management Problem and a Programmer's Opportunity", in *1981 National Computer Conference*, Vol. 50 of AFIPS Conference Proceedings, AFIPS Press, Alrington, VA, pp 343-347.

[2] Zamperoni, Andreas, Bart Gerritsen, and Bert Bril, *Evolutionary Software Development: An Experience Report on Technical and Strategic Requirements*, Leiden University, Leiden, The Netherlands, 1992.

[3] Hartrum, Thomas, and Robert Graham, Jr., "The AFIT Wide Spectrum Object Modeling Environment: An AWSOME Beginning", *in Proceedings of the IEEE 2000 National Aerospace & Electronics Conference (NAECON 2000)*, Oct 10-12, 2000, Dayton, OH, pp. 35-42.

[4] Hartrum, Thomas, and Robert Graham, Jr., *AWSOME Wide-Spectrum Language (AWL) Language Reference Manual*, Air Force Institute of Technology, Wright-Patterson AFB, OH, Jan. 2009

[5] Gamma, Erich, et al., *Design Patterns Elements of Reusable Object-Oriented Software*, Menlo Park, CA.: Addison-Wesley, 1999.

[6] Sarvepalli, Venkata (2003), *Automated Code Generation From a Formally Specified Class Definition,* M.S. thesis, Wright State University: U.S.

[7] Swamy, Sneha (2006), *Transformation of Object-Oriented Associations and Imbedded References to Them,* M.S. thesis, Wright State University: U.S.

[8] Shalloway, Alan, and James Trott, *Design Patterns Explained***,** Boston, MA: Pearson Education, Inc., 2005.

[9] Kuchana, Partha, *Software Architecture Design Patterns in Java*, Boca Raton, FL.:

CRC Press LLC, 2004.

[10] Bulka, Andy, "Design Pattern Automation", Aug 2003, *AndyPatterns Home*, retrieved Nov. 2009 from http://www.atug.com/andypatterns/AndyBulkaPattern %20Automation.pdf.

[11] Vlissides, John, et al., "Automatic Code Generation from Design Patterns", in *IBM Systems Journal*, Vol 35, Issue 2, 1996, pp. 151-171.

[12] Tsantalis, Nikolas, et al., "Design Pattern Detection using Similarity Scoring", in *Proceedings of the IEEE on Software Engineering*, Vol 32, Issue 11, Nov 2006, pp. 896-909.

[13] Gueheneuc, Yann-Gael, et al., "DeMIMA: a Multilayered Approach for Design Pattern Identification", in *Proceedings of the IEEE on Software Engineering*, Vol 34, Issue 5, Oct 2008, pp. 667-684.

[14] Singam, Arjun (2005), *Automated Code Generation from A Formally Specified Post-Condition,* M.S. thesis, Wright State University: U.S.

[15] Houle, Paul, "The Multiton Design Pattern", July 2009. *Generation 5*, Retrieved Nov. 2009 from http://gen5.info/q/2008/07/25/the-multiton-design-pattern/.