Automating the Application of Design Patterns based on UML Models

By:
Alexandre Sauvé

A thesis submitted to the Faculty of Graduate Studies and Research

In partial fulfillment of the requirements for the degree of

Master of Applied Science

Ottawa-Carleton Institute of Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

May 2005

Abstract

A software design pattern is a documented best practice solution that can be applied to a reoccurring problem; however there are often opportunities to apply design patterns which are overlooked by the designers of the software. This can be the result of inexperience, the sheer complexity of the system, or the fact that design patterns do not always constitute intuitive designs.

In this thesis, we present a structured methodology for semi-automating the detection of areas within a UML design of a software system that are good candidates for the use of design patterns. This is achieved by the definition of detection rules formalized using object-constraint language OCL and using a decision tree model. Presented in this thesis are example detection rules for a selection of design patterns from Erich Gamma's *Design Patterns*.

A prototype tool was developed to test the feasibility of this approach in practical situations. The tool was used to investigate the approach on two case study systems.

We conclude that the methodology produces positive results for the systems tested in the case study. We also discuss future work to ensure that the methodology is scalable and more user-friendly.

Acknowledgments

Firstly I would like to sincerely express my gratitude and thanks to Dr. Briand and Dr. Labiche for their help and guidance throughout my master's degree.

I would like to thank everyone at the SQUALL Lab for a great work environment.

I would like to thank IBM, NSERC and CITO for supporting me financially.

Lastly, but most importantly, I would thank my parents for their constant support throughout my studies.

Table of Contents

List of Figures

List of Tables

1 INTRODUCTION

Design patterns represent well-understood and established solutions to common OO design issues. Design patterns capture solutions that have evolved over time and are the result of experience and diligence as developers struggled for greater reuse and flexibility in their designs. Design patterns capture these solutions in a concise and easily applicable form. Since the release of the well acclaimed Design Patterns book by Erich Gamma et al. [26](also referred to as the Gang of Four/GoF patterns), design patterns have been on the fore-front of software research. Many tools have been developed to automate the implementation of design patterns [5, 44, 45]; however the user remains responsible for determining where design patterns can be used.

There exists many opportunities to make use of design patterns in modern software systems. But often there are situations where these opportunities are overlooked by the developer; this can be a result of inexperience, the sheer complexity of the system, or the fact that design patterns do not always constitute intuitive designs. Therefore, the usage of design patterns needs to be better supported and automated by a tool that would automatically provide, based on system static and dynamic information, some advice about where to apply design patterns. Such functionalities are already supported by tools [37], to some extent, but rely on the analysis of the source code. These tools thus entail that design decisions are already implemented, and require code refactoring which can be expensive. On the other hand, we are interested in providing tool support earlier during the development process, before any implementation is available. We are interested in supporting the identification of locations in UML [41] design documents (instead of source code) where design patterns could be applied. Working at the design level, instead of the source code level, would hopefully result in a more cost-effective use of design patterns. Alternatively the UML models can be reverse engineered from source code using modern case tools (as done in the case study of this thesis); therefore by using the UML model our approach can be applied to a variety of situations.

This work's main contribution is a structured methodology for automating the detection of areas in a UML design of a software system that are good candidates for the use of design patterns. By identifying areas where design patterns can be applied the software system is made more flexible, extensible, maintainable, portable and reusable [26]. In our approach we define detection rules that are able to identify those areas. They are precisely defined at a logical level on a simplification of the standard UML 2.0 meta-model, using Object Constraint Language (OCL) [40] expressions. Defining the rules at a logical level allows for the rules to be analyzed before their implementation and aids in the refinement of the rules after implementation. This work contributes example detection rules, namely for: the *Factory Method*, *Abstract Factory*, *Visitor*, *Adapter*, *State* and *Decorator* patterns [26]. The methodology can be used to create new rules and refine existing rules.

An Eclipse [14] based prototype framework has also been developed to test the feasibility of this approach in practical situations. The tool allows users to define new detection rules or edit existing ones using an intuitive interface. The detection rules can then be tested against UML models derived from existing systems. The example detection rules mentioned earlier were implemented using the prototype tool. These were then used to analyze two existing software systems: Elevator Control System [33] and ATM Test Driver [8]. Both systems were independently developed by fourth year graduate students. For both cases UML structures were found where the application of the suggested design pattern would improve the maintainability and extensibility of the software system. During the analysis of the two software systems a false positive[1] was detected by the tool, which led to further refinements of the detection rules. Therefore the approach suggested in this research has been proven effective in small scale examples to help developers identify areas to apply design patterns in existing systems.

---

[1] Defined as a suggestion made by the tool which should not be followed by the designer.

The thesis starts with a discussion of the latest research results in software design patterns (section 2). It then details our approach by using examples from the well known Gang of Four patterns (sections 3 to 8). For each of these patterns a brief description of the pattern and its applicability is given, followed by a description of the structure(s) upon which these patterns can be applied. The detection rules developed to identify these structures in the UML documentation are formally specified in OCL and an informal description is also provided. The information gathered from the UML diagrams to be used in the detection rules as well as any additional information required from the user will be itemized (section 9) then provides some details on how the tool supporting our strategy has been implemented. These detection rules will then be tested against available systems in an empirical case study (section 10). The thesis concludes with a discussion of the effectiveness of the approach and tools developed in aiding developers properly apply design patterns in real world applications (section 11).

2 STATE OF ART

What is a design pattern? A design pattern is a documented best practice solution to a given reoccurring problem applied in a variety of environments and situations [34]. Typically a design pattern aims to improve a system's flexibility, reuse, maintainability, and/or portability [26]. The design pattern will identify the key aspects of the proposed design structure and how those contribute to make it a reusable object-oriented design. The design pattern addresses the roles, responsibilities and collaborations that occur between the various components of the design. Finally the design pattern description will describe its applicability, the consequence and trade-offs to using the design pattern [26].

In this section of the thesis we address three important areas where the application of design patterns may require support and guidance:

1. Identifying areas where design patterns should be applied.
2. Applying the design pattern.
3. Ensuring that the design patterns that have been applied to the system are properly maintained in future revisions of that system.

As mentioned in the introduction, design patterns have received a lot of attention from the research community [10]. There are many texts and papers published describing design patterns and their uses, however with the possible exception of (2) the automation of the above issues have not been widely addressed in research.

This section of the thesis details the latest research in automated support for design patterns. It provides a structured and comprehensive survey where each of the three next sections focus in turn on the three issues discussed above.

2.1 Identifying locations to apply Design Patterns

The application of design patterns to a system begins with the identification of candidate areas to apply the design patterns. The decision to apply the pattern is very context based and requires knowledge of the pattern's structure and behaviour, the benefits and drawbacks, and knowledge of the system goals. There is limited research in the area of automation for determining candidate areas to apply design patterns.

2.1.1 Muraki et al. - Metrics for Applying GOF Design Patterns in Refactoring Processes.

Taichi Muraki et al. [37] have used software metrics to aid in determining areas in code where design patterns could be applied. To identify candidate areas, Muraki identified some poor design constructs or anti-patterns that should be replaced by design patterns. One of the poor design constructs that Muraki discusses is the use of conditional statements to perform the logic dictating the creation of various objects. This is illustrated in Figure 1.

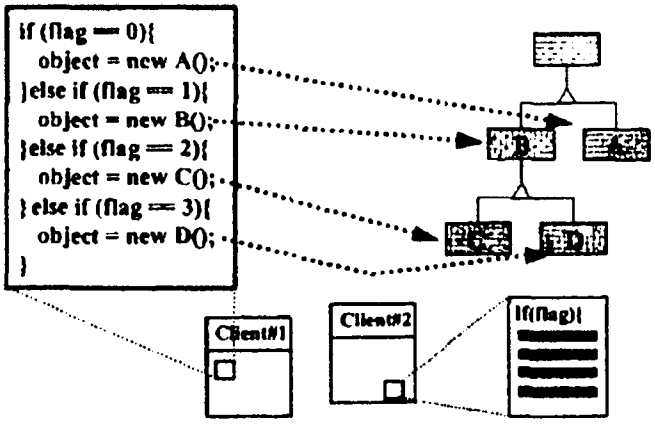

Figure 1 - Conditional Statements Governing the Creation of Objects [37]

The major problem with this design is extensibility. If a developer wishes to add a class to the system it could potentially require the modification of several areas in the code.

When adding a new class to the hierarchy structure presented in Figure 1, this would require that the developer modify all the conditional structures controlling the creation of this hierarchy, such as those presented in Client#1 and Client#2 in Figure 1. The use of some creational patterns such as *Factory Method* or *Prototype* pattern can improve this situation and provide a more extensible structure, since a new class can be added to the hierarchy with little modification of the client code. Figure 2 presents the application of the *Factory Method* to improve on the design in Figure 1. As seen in Figure 2, when a new class is introduced into the hierarchy a corresponding creator class is created that is responsible for the creation of the new class. The *Factory Method* localizes the changes required to the system when introducing new classes into the hierarchy.

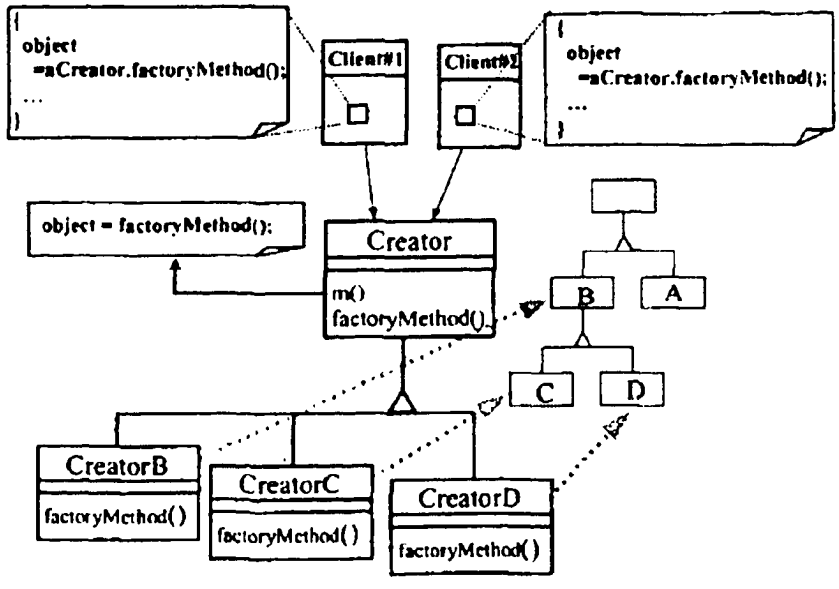

Figure 2 - Application of the *Factory Method* [37]

Another poor design construct that Muraki identified is related to inheritance structures, as illustrated in Figure 3. The problem arises when many subclasses override and invoke the same method in the super class. This creates a strong dependency between the subclasses and the super class making the alterations of the super class difficult. The

effort required to update or alter the super class and ensure proper functioning, makes this design less flexible.

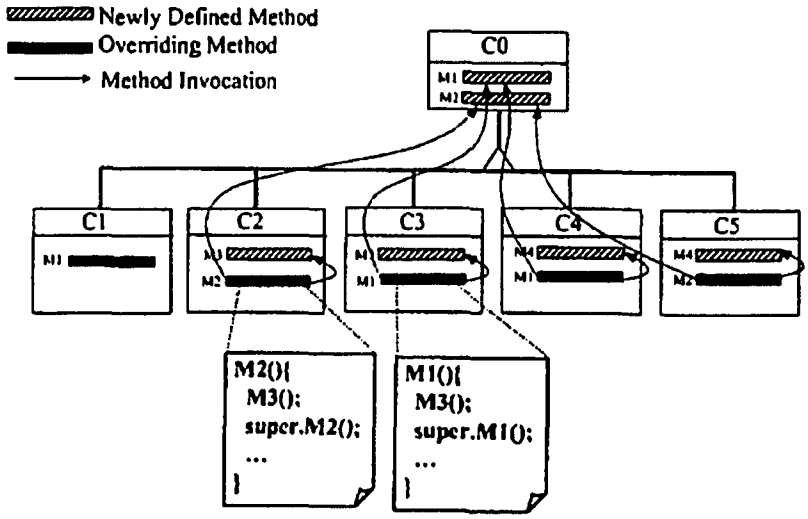

Figure 3 - Inflexible Inheritance Tree [37]

The use of the *Decorator* or the *Visitor* pattern will localize the functionality into a single class responsible for solely that functionality. This would facilitate changes to structure or behaviour of the inheritance tree. The application of the *Decorator* problem to the inflexible inheritance tree of Figure 3 is found in Figure 4.

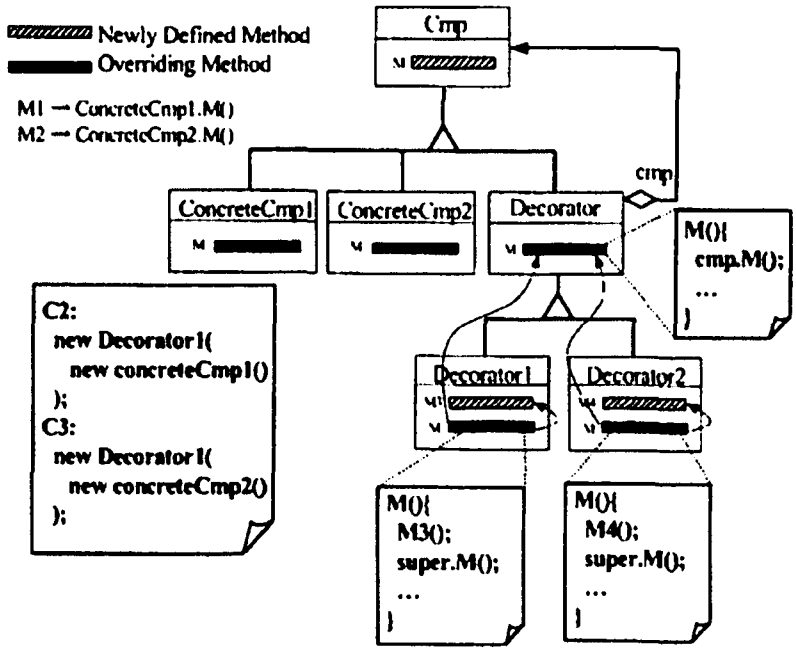

Figure 4 - Application of the *Decorator* Pattern [37]

Muraki et al. use code metrics to identify poor design structures such as those presented above. The metrics focus on conditional statements and inheritance structures. Some example metrics are presented in the list below.

CP1: the number of constructor statements appearing in a single conditional body.

CP3: the number of instantiated classes in the conditional body.

CP4: the number of inheritance trees that include the classes that can be instantiated inside a conditional body. For example, if a given conditional body instantiates classes A, B, C, & D and classes A and B are found in inheritance tree 1, while C and D are found in inheritance trees 2 and 3 respectively. The CP4 metric for that conditional body would be 3.

IC3: the total number of methods that override a method *and* that invoke the overridden method of the super class in the tree.

IM3: the number of classes that are instantiated by the methods that override a single given method. Another example to clarify this metric, if class A has method m1 and this method is overridden in classes B and C. The metric would equal the number of classes that are instantiated in method m1 of classes B and C.

The metrics are divided into 4 categories: metrics based on conditional statements and instantiation of objects (CP), conditional statements and method invocation (CM), based on inheritance trees (IC), and metrics based on overridden methods (IM). The authors try to map these metrics to design patterns, as seen in Figure 5, but this is rather loosely done. There is no discussion of how or why one given metric relates to a design pattern or a family of design patterns.

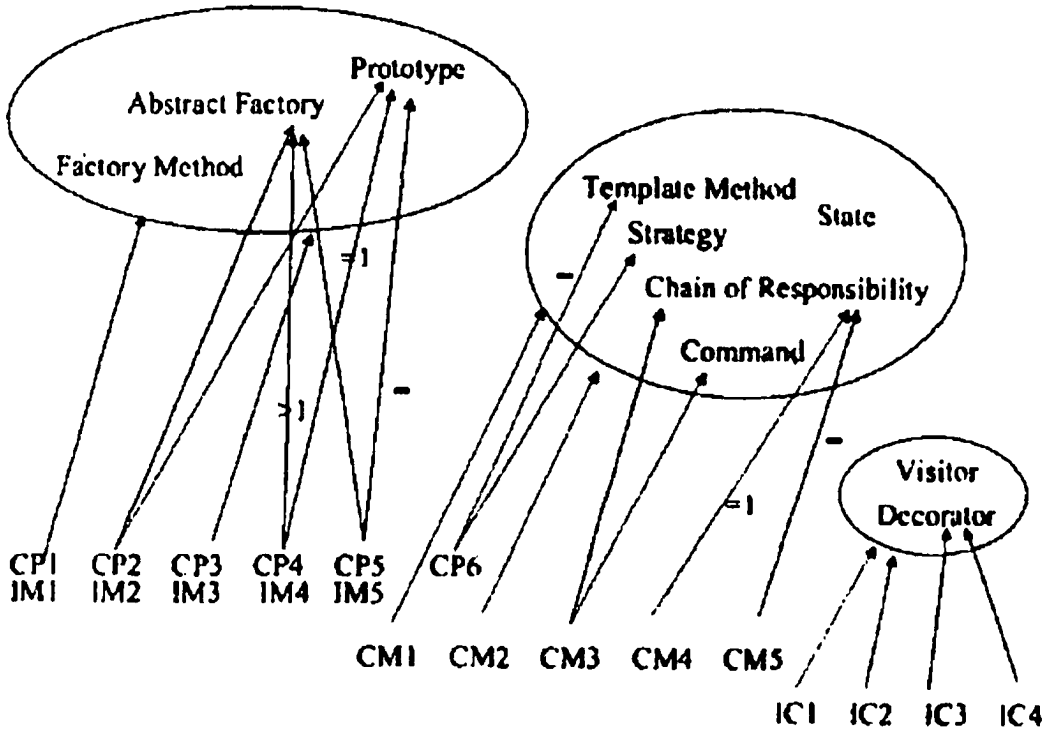

Figure 5 - Mapping Between the Metrics to the Design Patterns [37]

The only instruction given is that for the non-labeled arrows, the greater the metric measure associates to a stronger probability to apply the pattern or the pattern family. The arrows attached with the '-' means the lower the metrics measure suggests the use of the pattern. The Boolean conditions associated to arrows are used to recommend certain design patterns based on the metric. The authors do not mention how this mapping between the metrics was constructed, but it seems rather intuitive. For example, it seems intuitive to apply some form of creational pattern given that a large conditional body controls the instantiation of various objects given certain conditions. The metric CP4 is used to determine if the application of the *Abstract Factory* or the *Prototype* design pattern would be more appropriate, based on number of inheritance trees. This form of intuitive deduction can be performed for the remaining mappings.

2.1.2 Discussion

Muraki et al. present an interesting approach to identifying candidate areas where design patterns should be used. There is no automation for the improvement of the design; the research is solely to give the developer a suggestion on the application of a design pattern.

This research can be enhanced by guiding or suggesting to the developers on how to apply the design pattern to eliminate the anti-pattern. This would require a stronger mapping between the conditional statements and the corrective design pattern. At present a given metric could imply the use of several design patterns. In order to guide the developer the choice of design pattern would have to be more definite based on the metrics. This would mean a stronger mapping or a more refined mapping between the metrics and the design patterns. The present solution still relies strongly on the developer's knowledge of the patterns, and their ability to correctly choose the correct design pattern from the choices provided.

Empirical research is required to determine the commonality of the conditional statements mentioned in this research. As a case study the authors present a graphical

editor created by an undergraduate student. The graphical editor had undergone many revisions by the student and presented situations where Muraki's technique would identify poor design choices and the patterns to correct the problem. In order to validate this approach, more research is needed in determining the suitability of this approach on other software systems.

The precursors developed by O Cinnéide [38], also represent candidate areas to apply design patterns. This will be covered more in depth in section 2.2.2.

2.2 Automated Application of Design Patterns

The application of design patterns in a system involves a certain level of overhead, in design complexity and performance. In return the design is made more extensible, reusable, and/or flexible. Due to this trade-off, patterns are often not applied to systems in early stages of its lifecycle. It is impossible and impractical for the original designers to account for all the possible requirements that are to be placed on the system in the future. Future requirements may require the extensibility or flexibility that patterns provide. Therefore it is important that code and design be easily refactored to introduce design patterns when necessary. Many papers and research focus on the procedure for refactoring an existing system or the development of tools to automate the procedure. The highlights of this research are presented in this section.

Refactoring to introduce design patterns is a well-established practice in industry. There are many definitions for refactoring circulating in the software development community, however the most accepted definition

> ***Refactoring (noun):*** *a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. Martin Fowler [24]*

Refactoring is one of the corner stones of the XP software development process [4]. There is some notable literature on the subject of using refactoring to improve design, namely Martin Fowler's *Refactoring: Improve the Design of Existing Code* [24]. Joshua Kerievsky has extended Fowler's work in his book *Refactoring To Patterns* [32]. Although the text is still a draft, Kerievsky describes the methodology to refactor various poor designs to well accepted design patterns. As an example, Kerievsky describes the procedure for replacing hard-coded notifications with the *Observer* pattern. Both Fowler and Kerievsky present the motivation, the methodology, and an example for all the refactorings presented in their books. The discussion of the motivations will be interesting to all levels of design expertise; however the methodology and examples for the transformation are of little benefit to experienced designers.

Kerievsky also recognizes the need to refactor *away* from design patterns. Often design patterns are applied to a system in anticipation of future requirements that never materialize. In this case the design pattern is simply a performance hindrance to the system, and should be removed in favour of a more simplistic solution. For example, the *Observer* pattern has been applied initially to a single observer and subject in the anticipation that more observers will be added in subsequent releases of the software. If the requirements for that software change and no subsequent observers will be added then the application of the *Observer* pattern is excessive and should be refactored to a simpler solution.

2.2.1 Roberts - Eliminating Analysis in Refactoring

Numerous tools for refactoring have been developed, notably Don Robert's Smalltalk Refactoring Browser (SRB) [6, 45], which has been used by thousands of developers worldwide. The Smalltalk Refactoring Browser allows for the user to perform low-level refactorings which can be used to introduce design patterns. The process is not fully automated in that the user must identify the steps required to transform the existing system to one that uses the design pattern. A sample of the low-level refactorings provided by the SRB is presented below (for a complete list of refactorings available, please see [45]):

Add Class (*class, superclass, subclasses*)

This refactoring adds a new class to the system. This class is not referenced by other classes of the system, and can be inserted into the middle of class hierarchy.

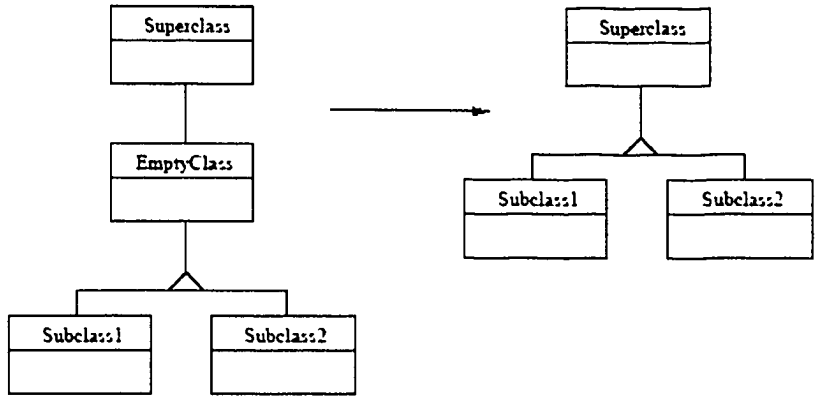

Figure 6 - AddClass Refactoring [45]

Add Variable (*class, varName, initClass*)

This refactoring adds an instance/class variable (*varName*) to a class (*class*). The variable is initialized to a given value (*initClass*).

Move Method (*class, selector, destVar, newSelector*)

This refactoring will move a given method (*selector*) from one class (*class*) to the other (*destVar*) under a new name (*newSelector*). The moved method is replaced by a forwarding method that invokes the method on the other class.

These alterations that the SRB makes to the software must be behaviour preserving in that the overall behavior of the software must remain constant as per the definition of refactoring presented earlier. To ensure behaviour preservation, each refactoring is coupled with a set of preconditions that must be met by the target program before the refactoring can be applied. In order to ensure that the program meets the preconditions, some analysis must be performed. To accomplish this Roberts defined a set of analysis functions used to validate the preconditions of the refactorings. The analysis functions are simply methods for interrogating the code to ensure that certain state. The analysis functions are separated into primitive analysis functions and derived analysis functions.

The derived analysis function can be computed using the primitive analysis functions. To illustrate the use of the analysis functions, the preconditions of the *Add Class* refactoring are presented in Figure 7, using the analysis functions developed by Roberts:

AddClass(*class*; *superclass*; *subclasses*)
pre: IsClass(*superclass*)
^¬IsClass(*class*)
^¬IsGlobal(*class*)
∀c∈ *subclasses*: (IsClass(c) ^ Superclass(c) = *superclass*)
post: IsClass' = IsClass[*class* / true]
ClassReferences' = ClassReferences[*class*/ϕ]
Superclass' = Superclass[*class*/*superclass*]
InstanceVariablesDefinedBy' = InstanceVariablesDefinedBy[*class*/ϕ]

Figure 7 - Pre and Post Conditions for the AddClass Refactoring [45]

The preconditions presented above ensure that:

- Exists a class named *superclass*
- No class exists named *class*
- There is no global variable in the system named *class*
- For all of the *subclasses*: the subclass does exist and it is subtyped from *superclass*

The postconditions indicate that after the refactoring the following is true:

- There exists a class named *class*
- There are no references to *class*
- *Class* is subtyped from *superclass*

- There are no instance variables defined in *class*

The low-level refactorings presented above are rarely applied by themselves, but rather in a larger sequence to make an effective design change. The analysis to ensure that all the preconditions are met for all the refactorings in a sequence is an expensive process in terms of computing. For the SRB to be an interactive tool, the processing for the analysis of the preconditions would have to be reduced.

For many refactorings, the conditions required to apply that refactoring have been arranged by the preceding refactorings; therefore the postconditions of the refactorings that have been applied to the program can be used to validate the preconditions refactorings to be applied. As an example, the application of the Strategy pattern in a program can be done with the following sequence of refactorings:

1. AddClass(*StrategyClass*;*Object*; *null*)
2. AddInstanceVariable(*Monolith*; *currentStrategy*; *StrategyClass*)
3. MoveInstanceVariable(*Monolith*; *foo*; *currentStrategy*)
4. MoveMethod(*class*; *methodName*; *currentStrategy*)

When evaluating the preconditions for this composite refactoring, the preconditions for the AddClass refactoring will be included; however for the other refactorings it has to be determined what preconditions need to be analyzed. Take for example the preconditions for the AddInstanceVariable refactoring, in the preconditions there is an assertion to ensure that the *StrategyClass* class does exist. This can be eliminated from the analysis as the postconditions of the AddClass refactoring specify the existence of the *StrategyClass* class. The analysis for the AddInstanceVariable refactoring can be reduced by comparing the preconditions with the postconditions of the AddClass refactoring. By applying this concept through the remaining refactorings, a set of preconditions can be deduced for the composite Strategy refactoring, as shown in Figure 8.

IsClass(*Object*)
^ ¬IsClass(*Strategy*)
^ ¬IsGlobal(*Strategy*)
^ IsClass(*Monolith*)
^ ¬HierarchyDefinesInstanceVariable(*Monolith*; *currentStrategy*)
^ ¬IsGlobal(*currentStrategy*)
^ DefinesSelector(*Monolith*; *implementor*)
^ ¬UnderstandsSelector(*Object*; *newImplementor*)
^ DefinesInstanceVariable(*Monolith*; *usedByImplementor*)
^ ¬DefinesInstanceVariable(*Object*; *usedByImplementor*)

Figure 8 - Precondition for Composite Strategy Refactoring

Performing this analysis of the postconditions can greatly reduce required analysis of the source code, making the SRB more interactive and user friendly.

Some preconditions cannot be easily verified by static analysis, or static analysis cannot provide an adequate approximation. In that case it may be necessary to use dynamic analysis to ensure that the system meets the precondition before applying the refactoring. There are also some cases where the use of dynamic analysis would reduce the time to validate the preconditions. The example presented by Roberts, is determining whether a given object is referenced by one other object at any given time (the object is exclusively owned by another object). Using static analysis to determine whether this precondition is met would only be an approximation. In order to ensure that a precondition is met the refactored code can be instrumented and exercised to verify that the precondition of the refactoring does hold. Roberts suggests using the method wrapper technique, in order to perform the dynamic analysis necessary validate preconditions. This technique presents code before and after methods or blocks of code of interest to monitor the variables during the block execution. Using the exclusivity precondition as an example, the instrumentation code would maintain a table of objects and number of references. If the object to be exclusively owned is determined to be referenced times through the use of the method wrappers, the precondition does not hold. If the instrumentation does detect

that the precondition does not hold then the refactoring must be undone. If the precondition does hold then the instrumentation can be removed.

In order to exercise the instrumentation the code must be thoroughly tested. The limitation of dynamic analysis is that in order to prove that a given characteristic of a program is true all possible control flows through that program must be exercised by a test suite. Only a small percentage of test suites provide this level of testing. As well the execution of the test suite can be time-consuming. Due to these limitations dynamic analysis can only provide an approximation as good as the test suite that it is coupled with. The combination of static and dynamic analysis may provide the best solution, however the technique to combine both the dynamic and static refactoring is not mentioned in this text.

All refactoring efforts have to be coupled with a powerful test suite for that program. Test suites should adequately test the specifications for a given program. By executing test suites on a given system a refactoring can be 'tested' to ensure behaviour preservation. If a program passes its test suite before the refactoring then it should pass after the refactoring has been applied. The ability to use a test suite to ensure behaviour preservation is proportional to the completeness of the test suite. Assuming that the test suite does adequately test the program against the specifications, would the dynamic analysis for behaviour preservation be required? Roberts does not discuss this topic in much detail, however suggests that dynamic analysis to ensure behaviour preservation has limited practicality.

Another interesting feature of the SRB is the ability to undo the refactorings applied to the system. This is done by associating an "undo" methodology to each refactoring, thus to undo a sequence of refactorings simply apply the associated "undo" actions in reverse order. This is a simple process and allows the developer to apply some changes to the system, inspect the overall effect, and if needed pursue a different course of action if the results are inadequate.

Many commercial IDEs [5, 14, 44] have adopted a subset of refactorings and features present in the Smalltalk Refactoring Browser. The use of low-level refactorings to introduce a high-level design pattern in an existing system is not a trivial matter. The texts mentioned previously could aid some inexperienced developers through the process; however the combination and application of the individual low-level refactorings is considered a manual process, prone to error, and is time consuming; a more elegant solution can be found.

2.2.2 O Cinnéide - Automated Application of Design Patterns: A Refactoring Approach

The most prominent work done in the area of automatically refactoring code to introduce high-level design pattern is by Mel O Cinnéide [38, 39]. The goal of O Cinnéide's work was to automate the introduction of design patterns into an existing system through refactorings. The refactorings that O Cinnéide developed are compositions of the lower-level refactorings similar to those developed by Roberts. Walter Zimmer [51] did very similar work in this area; however the published material on his research only outlines the initial concepts. Zimmer did not further elaborate on his work once O Cinnéide released his doctorate thesis. Other teams such as Eden [22, 23], Tokuda and Bantory [49, 50] worked on transformations that applied design patterns to code. These will be discussed later in sections 2.2.3 & 2.2.4.

Every refactoring O Cinnéide developed has a particular starting point, which he refers to as a precursor. The precursor is a construct found in the program from which the sequence of refactorings begins. To be useful this construct must be commonly found in software projects and provides in some manner the intent of the design pattern. The precursor for the *Factory Method* transformation is simply one class that creates a concrete instance of another class. This is illustrated in Figure 9.

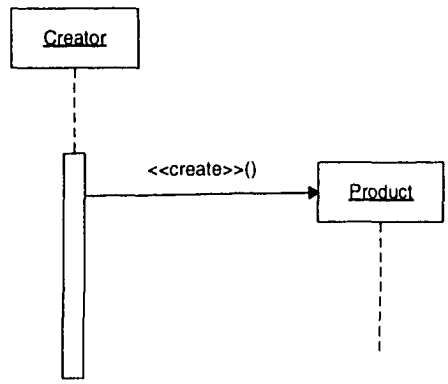

Figure 9 - Precursor for the *Factory Method* Transformation

A more complex example, the precursor for the *Abstract Factory* pattern, is a program that creates instances of classes found in a class hierarchy. The precursor structure is illustrated in Figure 10.

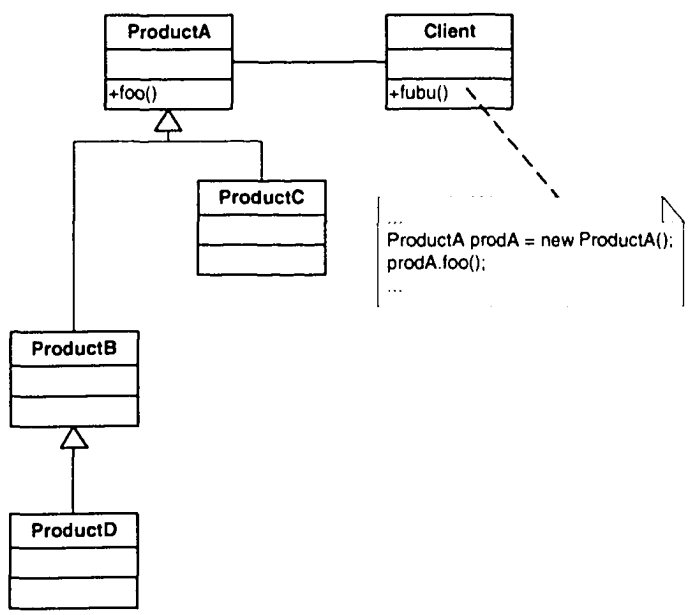

Figure 10 - Precursor for the *Abstract Factory* Transformation [38]

Once a valid precursor is determined for a pattern, the pattern is then decomposed into a set of minipatterns.

Minipatterns are lower level design constructs that can be combined to form a higher-level design patterns. Associated with the minipatterns are minitransformations, which apply the minipattern to an existing program. The minipatterns and minitransformations that O Cinnéide developed are listed below.

1. ABSTRACTION

The Abstraction minipattern adds an interface to a class that reflects how the class is used in some context. This enables another class to take a more abstract view of this class by accessing it via this new interface. The application of this transformation to the precursor of the *Factory Method* is illustrated in Figure 11.

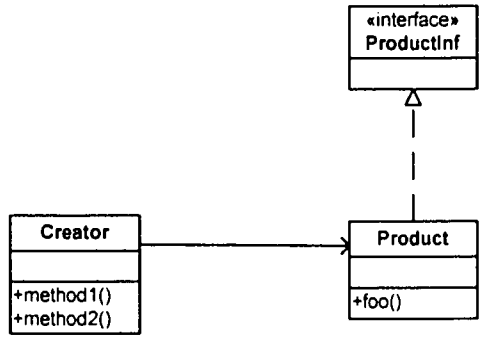

Figure 11 - Application of the Abstraction Minitransformation

2. ENCAPSULATECONSTRUCTION

The EncapsulateConstruction minipattern is applied when one class creates instances of another. The transformation weakens the binding between the two classes by packaging the object creation statements into dedicated methods. Figure 12 illustrates the

application of the EncapsulateConstruction minitransformation on the structure found in Figure 11.

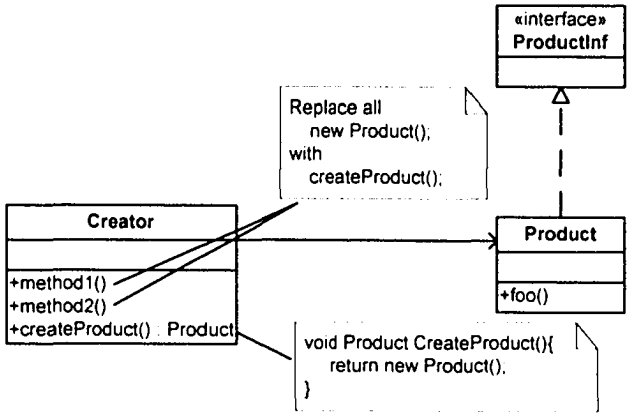

Figure 12 - Application of the EncapsulateConstruction Minitransformation [38]

3. ABSTRACTACCESS

The AbstractAccess minipattern is applied when one class uses, or has knowledge of, another class, and we want the relationship between the classes to operate in a more abstract fashion via an interface.

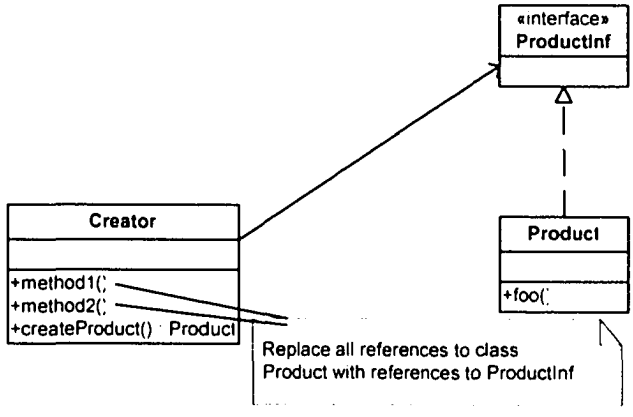

Figure 13 - Application of the AbstractAccess Minitransformation [38]

4. PARTIALABSTRACTION

The PartialAbstraction minitransformation constructs an abstract class from an existing class and creates an inheritance relationship between the two classes. This is related to the Abstraction minipattern, but instead of creating an interface for the class it creates an abstract class, this is why it is referred to as a partial abstraction. The abstract methods for the abstract class are specified for the transformation. The methods that are not specified to be abstract are pulled up into the new abstract class.

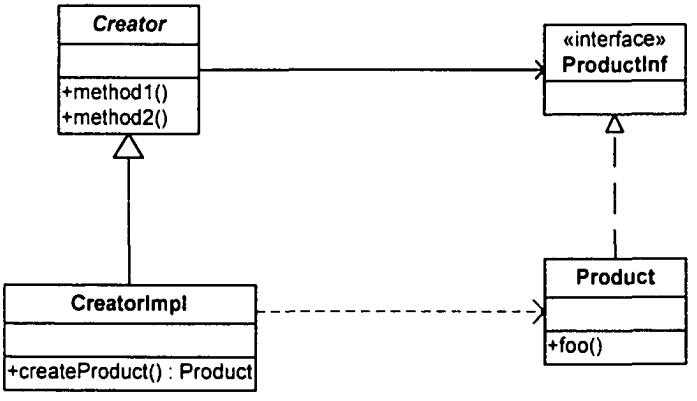

Figure 14 - The Application of the PartialAbstraction Minitransformation [38]

5. WRAPPER

The Wrapper minipattern is applied to "wrap" an existing receiver class with another class, in such a way that all requests to an object of the wrapper class are passed to the receiver object it wraps, and similarly any results of such requests are passed back by the wrapper object. This is similar to the Proxy pattern presented by Gamma et al.

6. DELEGATION

The Delegation minitransformation moves part of an existing class to a component class, and sets up a delegation relationship from the existing class to its component. The

transformation begins by creating an empty delegation class, and an exclusive relationship is created between the delegation class and the existing class (referred to as the context). The delegated method(s) are then moved to the delegate class. The SRB does a refactoring similar to this except that Roberts ensure that the method(s) to be delegated does not reference any instance variables from the context class or its inheritance tree. Roberts approach is too limiting for O Cinnéide's application. O Cinnéide's approach is to make the instance variable(s) referenced by the delegated method(s) public and to pass the context class as a parameter to the method(s). This approach seems to contradict the well accepted practice of making all data members of a class private and only accessible through set and get methods.

From the Figures 9-12 you can see the transformation to apply the *Factory Method* pattern to a program and how it is composed of minipatterns. In order to do the minitransformations associated with each of the minipatterns, lower-level refactorings, such as those in the SmallTalk Refactoring Browser, are used. For example, the Abstraction minitransformation is composed of several low-level transformations as show below.

```
Abstraction(Class c, String newName){
  Interface inf = abstractClass(c, newName);
  addInterface(inf);
  addImplementsLink(c, inf);
}
```

Figure 15 - Abstraction Minitransformation

Associated with each minitransformation is the precondition to ensure behaviour preservation and a postcondition to describe the state of the program after the refactoring. Again using the example of the Abstraction minitransformation, its preconditions and postconditions are presented in Figure 16.

precondition :
The class c exists :
isClass(c)
No class or interface with the name newName exists :
¬isClass(newName).¬isInterface(newName)
postcondition :
A new interface inf called newName exists :
nameOf = nameOf[inf /newName]
isInterface = isInterface[inf /true]
The class c and the interface inf have the same public interface :
equalInterface = equalInterface[(c,inf)/true]
An implements link exists from the class c to the interface inf :
implementsInterface = implementsInterface[(c,inf)/true]

Figure 16 - Preconditions and Postconditions for the Abstraction Minitransformation

Therefore each minitransformation is comprised of precondition(s), the algorithm for the transformation, and postcondition(s). For a high-level transformation consisting of several minitransformations, behaviour preservation can be proved by ensuring that all preconditions for the set of minitransformations be met before applying the larger transformation to the system. Using the same technique found in the SRB, O Cinnéide can reduce the analysis required to validate preconditions of a sequence of refactorings by comparing the preconditions of each refactoring against the postconditions of the preceding refactorings. In this manner O Cinnéide can derive a comprehensive set of preconditions for the high-level transformation to ensure behaviour preservation.

Returning to the example of the *Abstract Factory* pattern, the precursor for this high-level transformation can be found in Figure 10, an overview of the transformation algorithm is presented in Figure 17.

```
applyAbstractFactory(SetOfClass products, String newFactoryName, String newAbsFactoryName)
{
  addClass(createEmptyClass(newFactoryName));
  ForAll c : Class, c ∈ products {
    Abstraction(nameOf(c));
    AbstractAccess(allClasses, nameOf(c));
    EncapsulateConstruction(newFactoryName, nameOf(c));
  }
  applySingleton(newFactoryName, newAbsFactoryName);
  ForAll e : ObjCreationExprn, classCreated(e) ∈ products {
    replaceObjCreationWithMethInvocation(e, newAbsFactoryName + "getInstance().create" + classCreated(e));
  }
}
```

Figure 17 - *Abstract Factory* Algorithm

High-level transformations such as the *Abstract Factory* are not only composed of minitransformation and lower-level refactorings, but other high-level transformations. From Figure 17, it can be seen that the *Singleton* pattern (refer to text for more detail on this transformation) is applied to the newly created abstract factory class.

Analyzing the preconditions and postconditions of the individual refactorings that makeup the higher-level *Abstract Factory* pattern, O Cinnéide can decipher the preconditions necessary to ensure behaviour preservation before applying the pattern to a program. The preconditions for this transformation are presented in Figure 18.

precondition :
1. All the classes in products must exist, and for each class its interface name must not be in use :
$\forall c \in products \bullet isClass(c) \wedge$
$\neg isClass(interfaceName(c)) \wedge \neg isInterface(interfaceName(c)$
2. No class or interface may have the name *newFactoryName* or *newAbsFactoryName* :
$\neg$ isClass(newFactoryName) $\wedge \neg$ isInterface(newFactoryName).
$\neg$ isClass(newAbsFactoryName) $\wedge \neg$ isInterface(newAbsFactoryName)
3. The classes in products have no public fields :
$\forall f$: field, $\forall c$: Class, $f \in c, c \in products \bullet \neg$ isPublic(f)
4. The classes in products have no static methods :
$\forall m$: Method, $\forall c$: Class, $m \in c, c \in products \bullet \neg isStatic(m)$

Figure 18 - The Preconditions for the *Abstract Factory* Pattern

The overall result of applying this transformation to a program can be seen in the transition between Figure 10 and Figure 19.

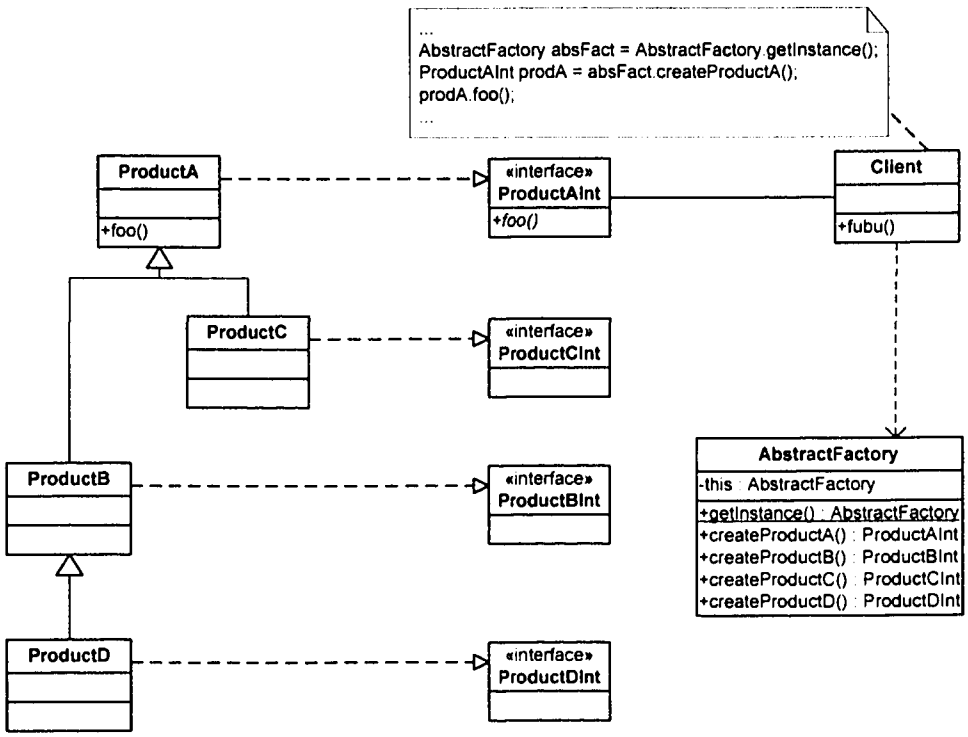

Figure 19 - Application of the *Abstract Factory* Pattern

This transformation can be very useful when extending a program to introduce a new family of products.

O Cinnéide was not able to produce such a transformation for all the patterns present in the Gamma et al. catalog. There are several reasons why:

- A convincing and useful precursor cannot be found in practice. Sometimes there is no compelling structure found in an existing program from which the transformation can be performed. In some cases there can be weaker precursors, where the intent of the design pattern is not apparent in the program. The example that O Cinnéide presents is the *Observer* pattern. A reasonable precursor for the *Observer* pattern would be a one-to-one relationship between a single subject and a single observer. There may be a requirement to add additional observers to the notifications, in which case the use of the *Observer* pattern would

be appropriate. The problem regarding the precursor is that the relationship between the two objects can be implemented in many fashions. This makes it difficult to construct a good transformation for this pattern.

- It may be difficult to recognize the precursor in an existing system. The main cause of this difficulty is that the precursor contains behavioural or dynamic elements that can be dispersed throughout the program and difficult to recognize through static analysis. This does not provide a solid starting point to perform the behaviour-preserving transformation.

- The transformation may still leave work for the programmer to perform in order to complete the application of the pattern. Depending on the amount of work left for the programmer the assessment for the transformation may vary.

Table 1 presents the Gamma et al. patterns and the ability to create an associated transformation using O Cinnéide's methodology.

Table 1 - Assesment of Design Pattern Transformations [38]

Pattern Name	Purpose	Assessment
Abstract Factory	creational	Excellent
Builder	creational	Excellent
Factory Method	creational	Excellent
Prototype	creational	Excellent
Singleton	creational	Excellent
Adapter	structural	Excellent
Bridge	structural	Excellent
Composite	structural	Excellent
Decorator	structural	Partial
Facade	structural	Impractical
Flyweight	structural	Impractical
Proxy	structural	Partial
Chain of Responsibility	behavioural	Excellent
Command	behavioural	Partial
Interpreter	behavioural	Impractical
Iterator	behavioural	Partial
Mediator	behavioural	Impractical
Memento	behavioural	Partial
Observer	behavioural	Impractical
State	behavioural	Partial
Strategy	behavioural	Excellent
Template Method	behavioural	Excellent
Visitor	behavioural	Impractical

For more information on the precursor and the analysis of the design pattern refer to the text. Note that O Cinnéide does not describe all the transformations listed in

Table 1; the following is a list of patterns that the transformation is described in detail:

- Factory Method
- Abstract Factory
- Builder
- Singleton
- Prototype

2.2.3 Eden et al. - Precise Specification and Automatic Application of Design Patterns

Eden et al. are [22, 23] doing similar work to O Cinnéide's, in that they introduce design patterns into existing systems using a metaprogramming approach [1]. The transformations are also organized in a similar manner, using the four levels of abstraction: design pattern, micro-pattern/minipattern, idioms/minirefactorings, and the abstract syntax tree. The differences between Eden's approach and that of O Cinnéide, is that Eden does not specify a precursor to the transformation and there is no mention of behaviour preservation for the transformation. The lack of a precursor and behaviour preservation allows for a developer to quickly alter or create design pattern transformations as needed. It also allows for a transformation to be applied under a number of various circumstances, not only those programs that the precursor can be found. The published material on this work is limited as Eden's main focus was in the development of LePus [22], which is another language for describing the behavioural and structural aspects of design patterns. The development of design pattern transformations was to validate the usefulness of LePus.

2.2.4 Tokuda and Batory - Evolving object-oriented designs with refactorings

Tokuda and Batory [49, 50] have developed a set of transformations that allow the user to apply design patterns directly to code. In their approach the developer applies the transformations as needed and must inspect the code to ensure that the transformation is proceeding as planned. This is similar to the Smalltalk Refactoring Browser, but geared

more to the application of design patterns to code rather than general refactoring. Note as well that Tokuda and Batory do not mention any aspect of behaviour preservation in their transformations.

2.2.5 Hamid et al. - Supporting Design by Pattern-based Transformations

All of the research mentioned previously is based on the application of design patterns to legacy systems. The transformations developed operation at the source code level. The research of Issam Hamid [30] of the Gelo group focuses on design pattern transformations for the design models. The premise of this work is to provide automation for the transition from high-level to low-level design. Figure 20 presents the transformation of a design to introduce the use of the *Observer* pattern.

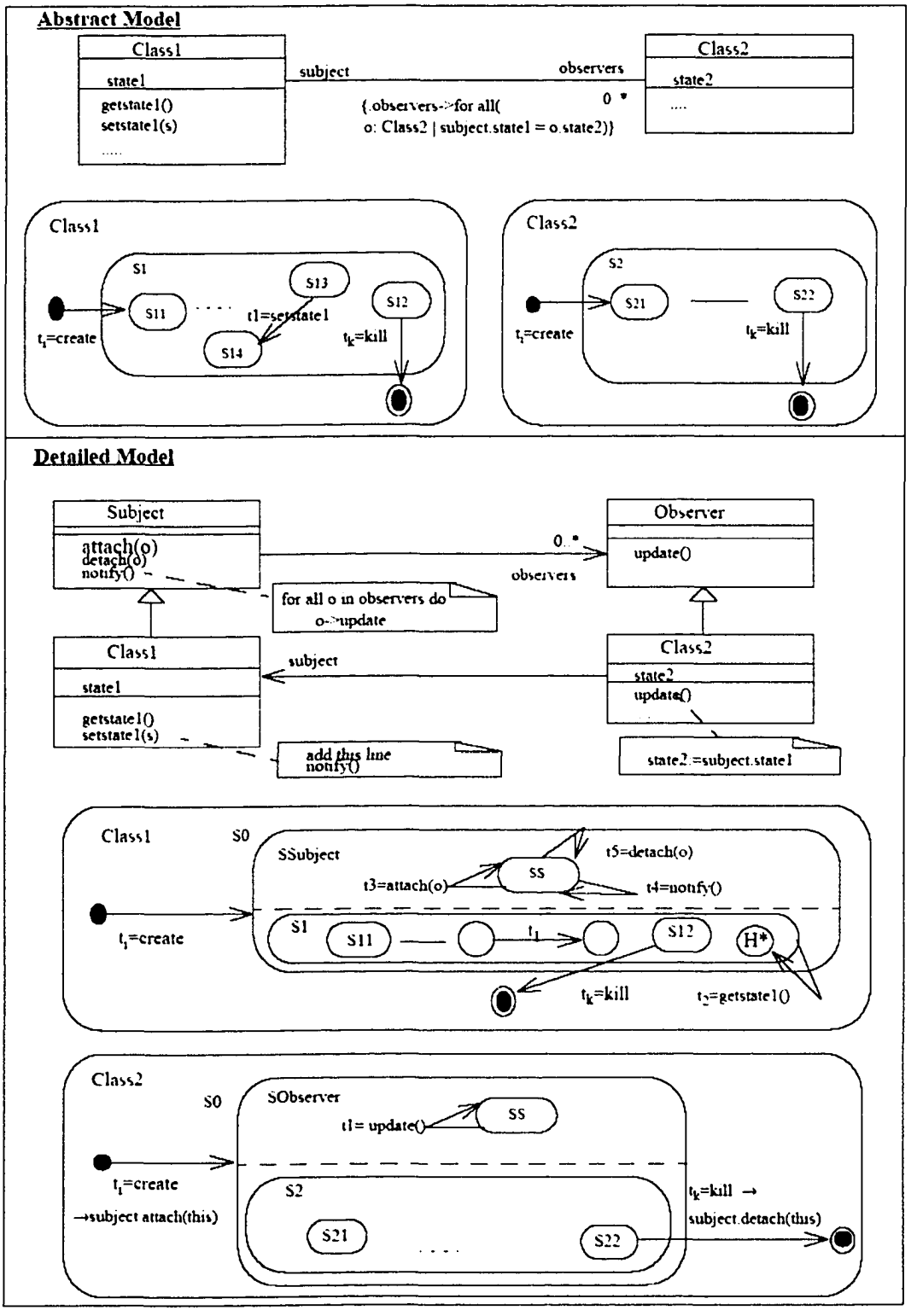

Figure 20 - Transformation for *Observer* Pattern

The transformations are composed of micro-transformations. Once the user selects a given element to apply a transformation, the tool will modify the class diagram, the state charts and the collaboration diagrams using the micro-refinements. This reduces the work for the developer and reduces the likelihood of errors. The tool will also keep track of the design patterns applied as a guide to the design choices made by the developers. The design patterns for which they have developed a set of transformations are: *Observer*, *Mediator*, *Façade*, *Proxy*, and *Forward-Receiver*.

2.2.6 Discussion

The Smalltalk Refactoring Browser has been used by thousands of developers worldwide, and has proven to be an industrial strength tool. The ability of the tool to provide generic refactorings to developer in a user-friendly manner is a great accomplishment. Some of the areas for future work are:

- The limitation was that it does not provide guidance to the application of the refactorings. This is somewhat touch upon through O Cinnéide's work, he presented the steps to apply high-level design patterns through refactoring. He described the starting point in the precursor, and described the smaller refactorings that could be used to introduce the design patterns.

- Refactoring a system in a multi-user environment. To provide an environment where multiple users can work independently on the system, refactor various components and the system's behaviour is preserved.

- As mentioned previously, the use of dynamic analysis in the validation of preconditions could be an interesting area of research. The current version of the SRB does not perform dynamic analysis. Before beginning this research some study on the practicality of dynamic analysis for behaviour preservation would be required.

The work done by O Cinnéide has several areas for future work:

- Only a limited number of the transformations mentioned by O Cinnéide were tested and there is little mention of the prototyping tool developed. O Cinnéide developed a tool, DPT, but there is little mention of its limitations or its ability to apply all the transformation mentioned. This thesis seems to be based more on theory than practical application; thus an interesting area of research would be to attempt to apply this research in a practical manner, and develop a strong industrial strength tool.

- To extend the catalog of transformations. There are several areas for this research: to develop transformations for those patterns that O Cinnéide found impractical, to develop alternative transformations for the practical patterns (various precursors for the same design pattern), or to look at other patterns not present in the Gamma et al catalog. In order to extend the catalog it may be necessary to handle various alternative situations that could arise in practice. For example the precursor for the *Observer* pattern was found impractical due to the multiple implementations of the dependency relationship between the subject and the observer. The development of transformations that can handle this variety in the code could be more practical than those developed by O Cinnéide, as it would loosen some of the preconditions and provide more options for the introduction of the design patterns.

- To maintain the pattern once the pattern is applied. This is similar in idea to the work done by Guéhéneuc. The system would have to monitor the changes made by the designer to ensure that all constraints associated with a given pattern are maintained. If the designer was to violate one of those constraints then they should be informed.

- To provide automation to determine if a given pattern should be applied to the system. This can be partially based on static analysis of the system, dynamic

analysis and user input on the goals of the system. This was mentioned in the discussion of the previous section. The precursors that O Cinnéide developed are obvious starting points for the application of patterns. Some analysis may be performed on the system to determine the candidate areas for the transformations, through the precursors O Cinnéide proposed, or other structures that relate to the intent of the design patterns.

- Often developers are over-zealous and apply patterns too liberally in software, or apply patterns that are never put to use due to changes in the system's requirements, contracts, etc. The ability to easily refactor away from a given design pattern can be very useful in practice.

When contrasting the approach of using low-level refactorings individually or creating compound high-level refactorings, it is easy to see that the low-level refactorings give the developer more flexibility. This popularity of the Smalltalk Refactoring Browser and similar refactoring features in commercial IDEs seems to suggest that developers tend to favour a greater control over the refactorings of a system. It would be interesting to see how a developer would react to a tool of high-level refactorings, which would make sweeping changes across the system. Perhaps a greater interaction with the user could alleviate some of the limitations found by O Cinnéide and others. Querying the user and having more conditional flows through the transformation may make it possible to create transformations for patterns that are impossible to fully automate?

The automation of the application of design patterns to design models as done by Hamid can be quite useful and time saving for designers if the automation is adequate. To better justify the work done by Hamid, more patterns would have to be observed and the result of the transformation on the design of the system. If the transformed design does not correspond to the design that a senior designer would produce then the automation would be futile. However, much like the transformations on the code level the transformations proposed by Hamid only need to provide the basic framework from which a design can expand.

Below a table is presented highlighting some of the differences between the works presented in this section:

Table 2 - Overview of Research Presented

Researcher	Language	Target level	Behaviour Preservation	Compound high-level refactorings
Roberts	Smalltalk	Source code	Y	N
O Cinnéide	Java	Source code	Y	Y
Eden et al.	Eiffel	Source code	N	Y
Tokuda and Batory	C++	Source code	N	N
Hamid	UML	Design model	N	Y

2.3 Maintaining Design Patterns

Once a designer has applied a given pattern to a program, it is important to ensure that the design pattern is applied properly to the system and that the structure of the design pattern remains intact over the revisions of the system. Inexperience with design patterns can result in poor implementation of a design pattern or distortion of existing design patterns.

2.3.1 Guéhéneuc et al. - Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-class Design Defects.

Yann-Gael Guéhéneuc et al. [28, 29] have developed a series of tools to help developers use and maintain design patterns. The tools developed are to support an integrated tool dubbed PTIDEJ (Pattern Trace Identification, Detection, and Enhancement for Java). The other tools developed are used as components in the PTIDEJ, but Guéhéneuc suggests they can be used as a standalone tool or integrated into other tools. These tools were developed for the Eclipse platform.

They have developed PDL (Pattern Description Language), which is a meta-model design explicitly for the instantiation and detection of design patterns. They use this meta-model to describe the structure of the design patterns, and it is also used as the intermediate form of the source code. Figure 21 presents a UML representation of the PDL meta-model.

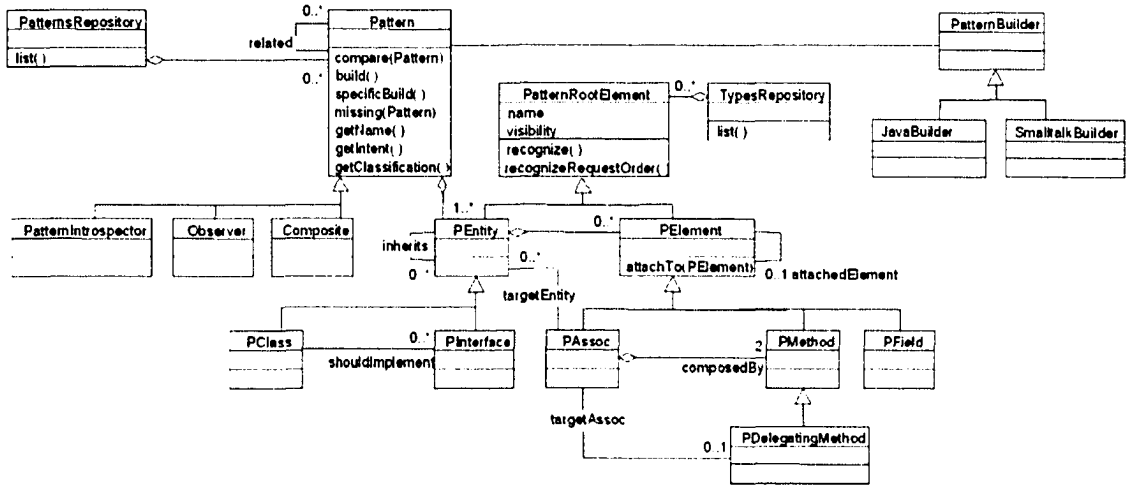

Figure 21 - PDL Meta-Model [29]

The model of a pattern would begin with an instance of the `Pattern` entity. The pattern would be composed of a collection of entities (represented by `PEntity`), representing the various classes and interfaces (represented in the meta-model by `PClass` and `PInterface`). Every entity has a collection of elements representing the methods, fields, associations, and delegations (represented by `PField`, `PMethod`, `PAssoc`, and `MDelegatingMethod`) for that entity. An important note of the meta-model is that it models only the structural aspects of the design pattern, none of the creational or behavioural aspects are taken into account (with the exception of the delegate method).

Based on the meta-model representation of the design pattern a series of constraints are formulated against the composite elements of that design pattern. There is no mention of the procedure used to develop these constraints from the meta-model version of the pattern; however Guéhéneuc does mention the derivation of the constraints is a manual process. In order to recognize design patterns from the transformed source code, a constraint solver is used, PALM (Propagate and Learn with Move).

PALM is a constraint solver with enhancements geared to this application, in that PALM provides a means to identify which set of constraints is missing from a given set of entities. This allows the system to automatically discover distorted forms of the design

pattern based on the precise form, by removing specific constraints until the distorted pattern is found. Each of the distorted design patterns discovered are associated with the set of constraints that were relaxed to discover that structure in the code.

Once an imprecise design pattern is found, Guéhéneuc provides the means to transform the distorted pattern to the correct form specified in PDL. In order to perform the transformation, each constraint for a given design pattern is associated with a transformation which will enforce the constraint. By applying transformations for the constraints that were relaxed by PALM to discover the distorted pattern, the source code can be modified to comply with the pure design pattern. This transformation is done using JavaXL, which is simply a source-to-source transformation tool for Java.

There is limited discussion of the application of this approach to industrial systems. The only example design pattern presented by Guéhéneuc is the *Composite* pattern. This pattern would be the ideal candidate for this constraint-based system, as the pattern contains several associations and inheritance between objects. Judging by the PDL meta-model presented in Figure 21 and the classes which subtype the `Pattern` entity, Guéhéneuc only applied the meta-model to the *Observer* and *Composite* design patterns.

2.3.2 Discussion

There are several limitations of this system that Guéhéneuc addresses:

1. The meta-model they propose only focuses on the structural aspects of the design pattern (with the exception of the delegate method). This limits the design patterns to those that are structural in nature, and ignores the behavioural and creational design patterns.

2. Constraints used to detect a design pattern have to be entered manually. As future work, Guéhéneuc hopes to generate the constraints based on the meta-model of the design patterns.

3. As with most design pattern recognition tools, the PTIDEJ has difficulty dealing with the dynamic aspects of a software program. There is a limit to the knowledge the tool can collect from static analysis of the source code and express in the meta-model. This limits the ability of the tool in the expressiveness of the meta-model and in the automation of the detection process.

4. They have scalability issues in mapping their meta-model over large applications.

5. Depending on the language used not all the constraints will have associated transformations. The difficulty lies in the fact that the constraints are expressed using the meta-model constituents and the transformations are applied at the source code level. This gap between source code and the meta-model, presents a gap between the transformation and the constraints.

Apart from the limitations presented above I believe that this solution presents many drawbacks not addressed by Guéhéneuc.

1. It is a rather large assumption to assume that a solution that is dissimilar to a design pattern in some minor fashion should be considered a design defect. Many solutions will deviate from the design pattern for valid reasons based on the context of the application and the goals of the application. However, if this is the case the designer can choose to ignore the advice given by PTIDEJ.

2. The transformations are not behaviour preserving, therefore applying the transformation in the hopes of improving the design will most likely break a working solution. Even worse the transformations applied by JavaXL are guaranteed to be syntactically correct, but cannot ensure that the transformations are semantically correct. Therefore after the transformation, a working program may not even compile. When applying the transformations in an existing system, this would not change the underlying algorithms; therefore PTIDEJ would change the structure of the system, but all the structural changes would not be utilized. It would be up to the developer to ensure that the changes made by PTIDEJ would

be used, PTIDEJ attempt to aid in this regard by providing descriptions of the design patterns and the intended behaviour.

Another approach to this problem would be to recognize design patterns present in programs and then ensure that the pattern structure remains intact. A tool can be developed that would recognize correct design patterns and monitor the changes that a user makes to elements of that design pattern in order to ensure that the correct structure remains. The tool could also guide the user in adding new elements to the design pattern and ensure that the design structure is maintained. The benefit of this approach is that the tool could detect the erroneous changes early and inform the user early before he/she makes a large number of changes to the pattern structure. The early detection would aid in restoring the correct design pattern. The tool would act as a learning tool for those developers not familiar with design patterns or those not familiar with particular design patterns.

This capability could be integrated into IDE, but could also be a useful feature to code source repositories, such as Rational's ClearCase [43] or CVS [25]. The repository could analyze the changes that a designer has made to the source code and ensure that all design patterns remain intact before allowing the change to be checked in.

3 APPROACH

The previous section of the thesis provides a comprehensive survey of the latest research in the area of design patterns. As mentioned in the introduction the application of design patterns requires automated support and guidance in three areas:

1. Identifying areas where design patterns should be applied.
2. Applying the design pattern.
3. Ensuring that the design patterns that have been applied to the system are properly maintained in future revisions of that system.

As seen in the previous section, the majority of the recent research has been in the automation of (2). However in the area of identifying candidate areas to apply design patterns or maintaining design patterns that have been applied to the system, the recent research is somewhat limited. We believe that automated support and guidance for design patterns can help the systematic and precise application of design patterns into a software systems design. Through the correct use of design patterns the ease of maintenance, extensibility and flexibility of the software are greatly improved.

Many people in the pattern community do not believe in automated support for the application of design patterns. The following quote by James Coplien summarizes this position:

> Patterns aren't designed to be executed or analyzed by computers, as one might imagine to be true for rules: patterns are to be executed by architects with insight, taste, experience, and a sense of aesthetics. [11]

This may be one of the key reasons for the lack of research in this area; however, through the use of precursors [38], O Cinnéide formalized structures on which design patterns could potentially be applied. He then developed transformations to apply design patterns

to software programs. The notion of the precursor or the structure on which a design pattern may be applied is at the center of this research. O Cinnéide's work clearly demonstrates the feasibility of automating the use of design patterns; however in credit to Coplien's statement, we believe the designer should always be involved in such design decisions.

The aim of this research is to provide semi-automated identification of areas in the UML design where software design patterns could be applied. This can be achieved through the identification UML constructs that are strong candidate for the application of a design pattern (similar to the notion of a precursor introduced in O Cinnéide's work). To identify these structures analysis of the UML artifacts must be performed in order to help the designer determine where the design pattern should be applied. There are two key advantages to using the UML design in performing such analysis.

First UML design documents are available early in the design stages of development. In contrast, most of the research presented in this report was based on using the static code analysis of an existing system. This limited the methodologies developed to be only applicable after the code for the program was generated. Our intent is to provide guidance of the application of patterns at the design stage, before the implementation of the system. In order to perform this analysis we must make use of the UML artifacts generated during the design, structural and behavioral diagrams, as well as OCL constraints where required. The early guidance for developers is crucial in that it reduces the effort required to integrate the design pattern into the design of the system. However for existing systems modern tools, such as Together [5] or Rational Software Architect [44], provide a means of generating UML documentation from the code base of a software system. Though this documentation is not always complete, it can provide enough information to be used for the analysis of the design of a system. This is shown in the case study section of this thesis.

Secondly, the analysis of complete UML diagrams can yield much more information about the system, than the static analysis of the code. UML artifacts contain information

regarding the functional specifications of the system (use case diagrams), the dynamic behavior of the system (interaction diagrams), and its structural information (deployment and class diagrams). A great deal of information present in this UML diagrams and associated OCL constraints cannot be easily and fully extracted from the code using static or dynamic analysis. For example, in section 2.2.1, the use of dynamic analysis was discussed in the context of the Smalltalk Refactoring Browser [45]. Robert proposed the use of dynamic analysis to determine certain characteristics of the software that could not be deduced using static analysis of the code. The example presented by Roberts was the exclusivity of an object. Roberts indicated that in order to determine if an object is exclusively owned by another object requires dynamic analysis of the system, as static analysis can only provide an approximation. To perform the dynamic analysis requires the use of complicated and expensive methods to adequately exercise the system. As well the result obtained using dynamic analysis may remain inconclusive. In comparison, UML diagrams can provide further insight on the dynamic behavior of the system. The exclusive ownership of an object is clearly expressed on the class diagram (given that the system does conform to the diagram) as a composition association between the two objects. It is our belief that using the UML artifacts, a better analysis of the system may be performed.

The research projects presented in section 2 always aimed at a fully automated approach to using design patterns; however, as mentioned earlier, the decision of when and where to use design patterns is highly context sensitive. The contextual information required to properly apply a software design pattern cannot be extracted from the UML model; therefore the tool built interacts with the user in order to obtain the required information. Part of the research performed was determining what information would be required from the user to accurately suggest the use of a design pattern. An important note is that the input from the user is part of the overall analysis and in some cases results in further automated analysis of the UML diagrams, potentially leading to levels of analysis to be performed before any conclusion can be made. The analysis methodology developed is

represented as a directed decision tree with nodes representing various analysis steps, and the leaves representing the suggested design patterns, this is illustrated in the Figure 22.

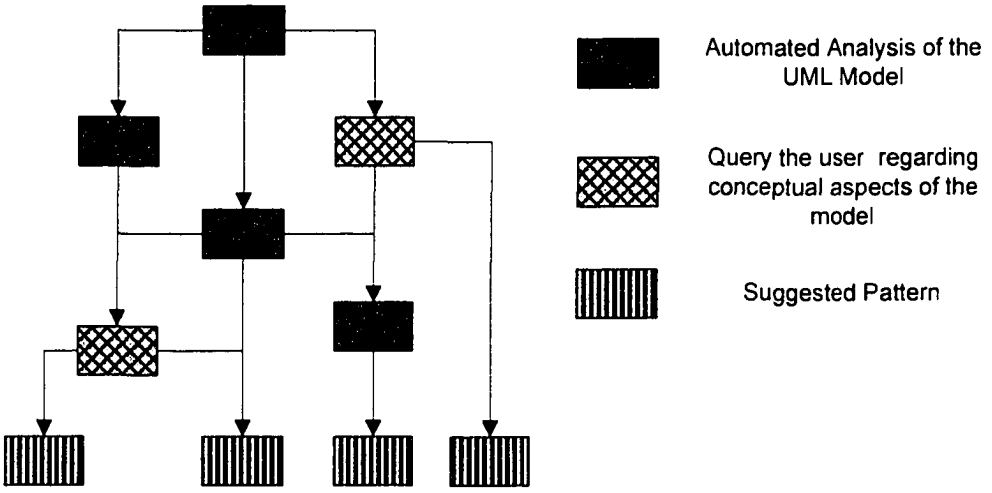

Figure 22 - Decision Tree to Determine Proper Design Pattern

The automated analysis to be performed can be elegantly expressed using OCL against the UML 2.0 meta-model [41]. The branches exiting a node represent various conditions to be applied on the result generated from the analysis performed at that node. These conditions will be used to ascertain the next step of analysis to be performed or a particular design pattern to be applied. The main research path in this work was the specification of decision trees that can be used to give constructive suggestions to the developer. The process is an iterative approach where further refinements of the decision trees are to be expected, as more empirical findings become available.

The use of the decision tree and accompanying OCL constraints for the specification of the analysis provides an elegant solution as the decision tree and its navigation can be changed independently from the analysis performed at each node. The OCL provides a logical specification for the analysis; but as the tooling developed also supports OCL (see section 9.1.2), it can be used to implement the analysis in a more concise and clear form in comparison to Java. The readability and conciseness of the OCL, as well as the navigation through the decision tree allow for the decision trees developed to be flexible

and easily modified: by altering or deleting existing nodes, modifying the navigation between nodes or defining new nodes to be added to the decision tree.

An important note is that the structures and the analysis used in the decision tree could potentially target anti-patterns in the design. If an anti-pattern is found the tool should then proceed to determine a good solution to the problem based on further analysis or user input. This would be similar to the research done by Muraki et al. (presented in section 2.1.1) with the exception that our research focuses on the design model level of abstraction and exploits the UML models at our disposal.

Sections 4 through 8 describe the decision tress generated for a subset of the popular Gang of Four patterns [26]. These sections present the specification of the decision trees from which the trees can be implemented in the DPATool (discussed in the Implementation section of the thesis).

4 FACTORY METHOD/ABSTRACT FACTORY PATTERNS

4.1 Factory Method Pattern

4.1.1 Description

The *Factory Method* pattern separates the code that will make use of a product object from the code that will create the product object. The structure of the *Factory Method* pattern is shown in Figure 23.

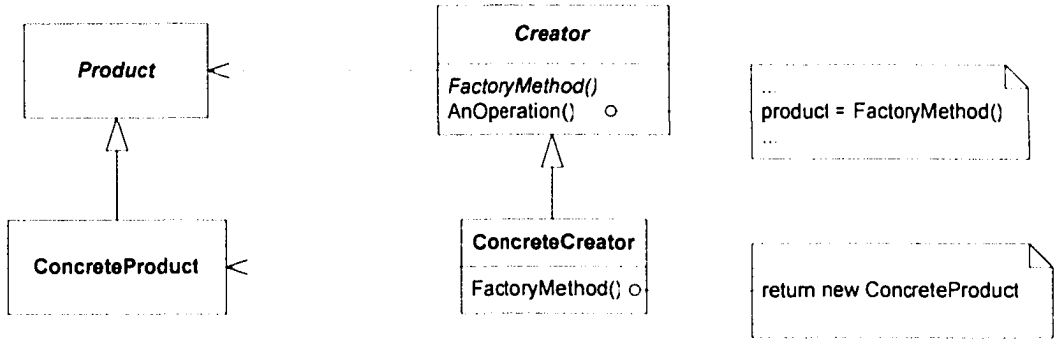

Figure 23 - *Factory Method* Pattern Structure. [26]

4.1.2 Applicability

As mentioned in [26] the *Factory Method* is applicable in a variety of situations:

- When the client cannot anticipate the class of objects it must create.
- A class that wishes to delegate the creation of objects to its subclasses.

The pattern allows for the client to substitute the concrete product to suit their needs. It allows for more classes to be easily introduced into the hierarchies, as the pattern isolates the code changes necessary to a single function that is easily overridden using inheritance. If such extensibility of the hierarchy is not required then the use of the

design pattern may be superfluous. This is a decision that must be made by the designer of the system with consideration for the future uses/needs of the system. The *Factory Method* does incur some overhead with the required method call. In most situations the overhead of an extra method call would be considered negligible, but there are situations that could arise where performance is of utmost importance. Most design patterns incur some performance overhead; it is the responsibility of the developer to weigh the trade-off between performance and maintainability.

4.1.3 Discussion of Pattern Candidate Structures

4.1.3.1 The Candidate Structures

A 'pattern candidate structure' or 'candidate structure' can be seen as a situation present in the software design where a design pattern could be applied but was not. This section of the thesis gives textual descriptions of such structures and justifications as to why they are considered pattern candidate structures for the *Factory Method* pattern. As being a creational pattern, the main situation that we are interested in is where the client instantiates the concrete classes directly, this is illustrated in Figure 24.

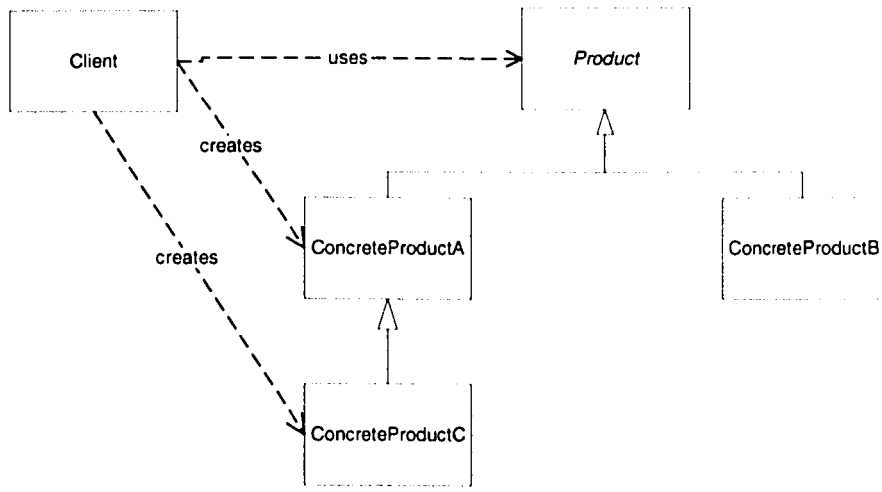

Figure 24 - Basic Candidate Structure for the *Factory Method* Pattern

Figure 24 shows the basic candidate structure for the *Factory Method* pattern. The design illustrated here limits the extensibility of the application. If the client creates objects of the subclasses of `Product` in many different locations in the code, or if the class type created is likely to change then the choice to use a design pattern is further justified. In this case, the use of the pattern is warranted as it reduces the overhead of adding new classes into the hierarchy, as the code change would be centralized and not spread throughout the software system.

If the `Product` hierarchy is not expected to change and if very few creation statements are visible in the `Client`, then the use of the design pattern may not be required, but this is an unlikely situation and such cases should nevertheless be conveyed to the designer. The decision of using the pattern is then up to the designer of the system with consideration for the future uses/needs of the system. The *Factory Method* pattern introduces some overhead to the creation of the product objects through additional method calls and this is a trade-off between performance and maintainability that must be considered.

The information in Figure 24 is an indication that the *Factory Method* pattern may be useful but if we manage to obtain more precise information about how the `Client` is creating `Product` instances, we are able to be more precise in our recommendations to the user. Figure 25 shows a more specific pattern candidate that indicates more clearly a stronger need for the *Factory Method* pattern than Figure 24. We clearly see that different conditions lead to the creation of instances of different `Product` concrete subclasses. If we can further determine that such a creation scheme is present in several operations of `Client`, then the use of the *Factory Method* pattern is even more justified. Such information can be extracted from UML interaction diagrams and/or OCL postconditions for the `Client` operations.

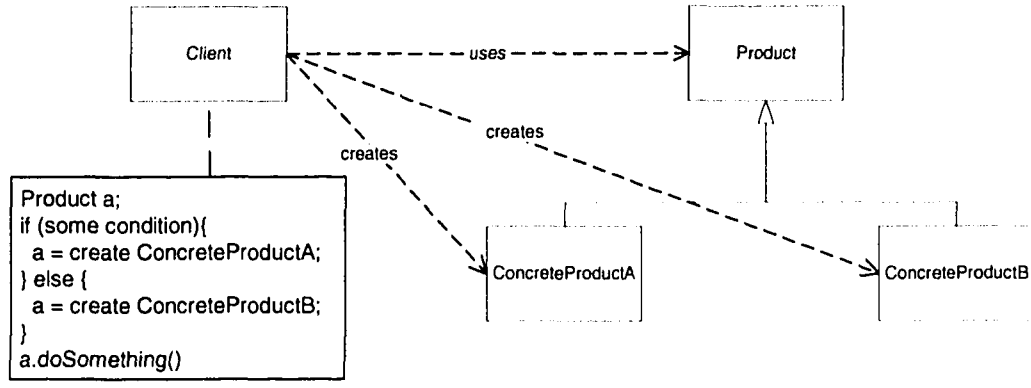

Figure 25 - A Stronger Candidate Structure for the *Factory Method* Pattern

If a client uses an alternative combined fragment to choose which sibling classes (`ConcreteProductA` and `ConcreteProductB`) to instantiate this suggests that the siblings are implementation classes with closely related responsibilities. In that case, the parent class (`Product`) will likely provide the interface used by the client and we can expect additional concrete classes to be introduced in the future when the system is extended. This is similar to the anti-patterns discussed in the research of Muraki et al. [37] presented in section 2.1.1 and is similar to the precursor developed by O Cinnéide [38].

4.1.3.2 Heuristics to Analyze UML Diagrams

Given the above discussion, what kind of search heuristics should we follow to determine candidate structures? Our goal is to develop heuristics to quickly reduce the search space and enable reasonable response-time even within large models. One way to do so is to start by identifying interesting inheritance hierarchies from the UML class diagram. These hierarchies and their clients can then be further investigated.

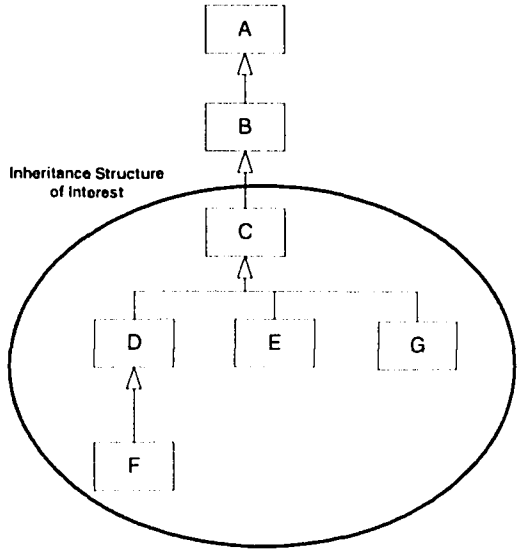

Figure 26 - Inheritance Tree

As illustrated in Figure 26, we have to pay particular attention to cases where numerous subclasses can be found on the same hierarchy level. However, the commonality of such structures may make it a poor choice to reduce the search space initially as it may lead to many inheritance hierarchies being selected. In each case, the clients that warrant further investigation are those that instantiate concrete subclasses: Assuming A, B, and C are abstract, we would consider clients instantiating objects of type D, E, F or G in Figure 26. These objects can be declared of type C, B or A in the client code, and therefore this would result in associations or dependencies between the client and those classes in the class diagram. There are however many possible configurations for clients and inheritance hierarchies such as the one in Figure 26. This process of looking at the inheritance hierarchies can only guide the search as to which clients to investigate first. Furthermore, this search is complicated by the fact that there might be future classes that will be added to the inheritance hierarchies and that we may not be aware of. This information can only be obtained from the class diagram if the "incomplete" stereotype is used or if we ask the designer.

Another source of variation is that inheritance hierarchies may not be only class based, but they may make use of interfaces. Such cases need also to be accounted for in our analysis.

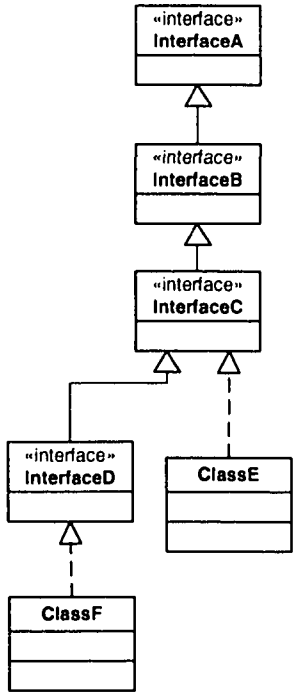

Figure 27 - Tree Built Using Interfaces

Figure 27 shows an example where interfaces are used instead of concrete or abstract superclasses. We have to take into account hierarchies that are composed of both classes and interfaces. Since abstract classes or interfaces cannot be instantiated; their clients should be highly scrutinized as the use of creational patterns is likely to be warranted. Indeed, the presence of an interface/abstract class suggests that many different implementations/subclasses will be created even though this may not be the present situation.

Another alternative heuristic is to begin the search for candidate structures with the conditional creation of objects in sequence diagrams. This is the approach taken in the detection rules presented in this report. The conditional creation of objects can be found on sequence diagrams and an example of such a sequence diagram can be found in Figure

28. Figure 28 also describes the concepts of 'alternative combined fragment' and 'interaction operands' as discussed in [41]. We can observe the creation of objects that share a common interface within an alternative combined fragment and the client that interacts with a common generalization of the newly created object.

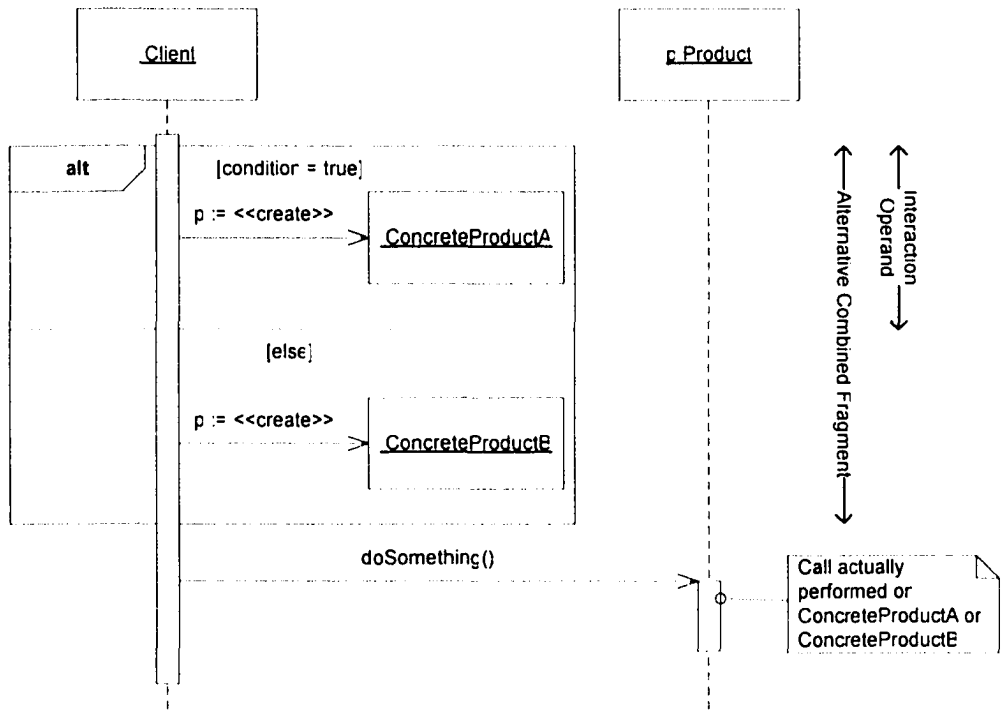

Figure 28 - Sequence Diagram Indicating Conditional Creation

Sequence diagrams such as the one in Figure 28 can either be the result of the analysis and design process or can be the result of some reverse-engineering effort. The source of the diagrams will not be discussed in this research.

By presenting alternative heuristics, this illustrates that using our methodology to identify candidate structures can result in various algorithms being developed.

4.1.3.3 Alternative Candidate Structures

One aspect that complicates the search is the many alternative ways of expressing the same information or very similar information in a sequence diagram or any UML

diagram for that matter. For example an alternative candidate structure is presented in Figure 29. The introduction of the loop structure is only a minor alteration to the candidate structure, but enough of a change that if not taken directly into consideration it will not be identified by the detection rules. The OCL rules created at this stage (presented in section 4.3.2) will not detect this candidate structure.

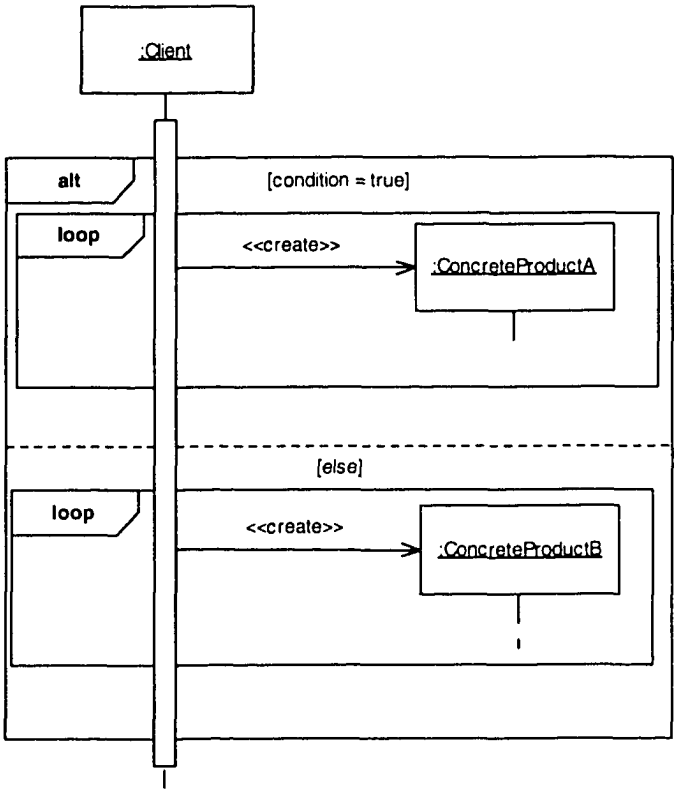

Figure 29 - Sequence Diagram with Loop within Alternative

It is important to determine which are the more common candidate structures and alternative structures as to develop OCL based detection rules that are general enough for detecting these structures. The rules developed in section 4.3.2 can be generalized to include the alternative candidate structure presented in Figure 29; however this may not always be practical based on the commonality of this structure. However it is also important to refine the OCL rules to ensure that they detect the candidate structures that are often found in software systems. There may be several sets of rules developed for a

single pattern as to adequately identify the alternative candidate structures that are considered of high importance. These rules will be constantly refined based on empirical findings. Therefore any tool that is developed to implement our methodology must have the capability to easily extend/modify the set of OCL detection rules used to identify candidate structures.

4.2 Abstract Factory Design Pattern

4.2.1 Description

The *Abstract Factory* design pattern is used to provide an interface to creating families of related objects without having explicit knowledge of the implementation classes. This is achieved by creating a factory for creating these objects. As seen in Figure 30 below the `AbstractFactory` class provides methods for creating the objects required by the client.

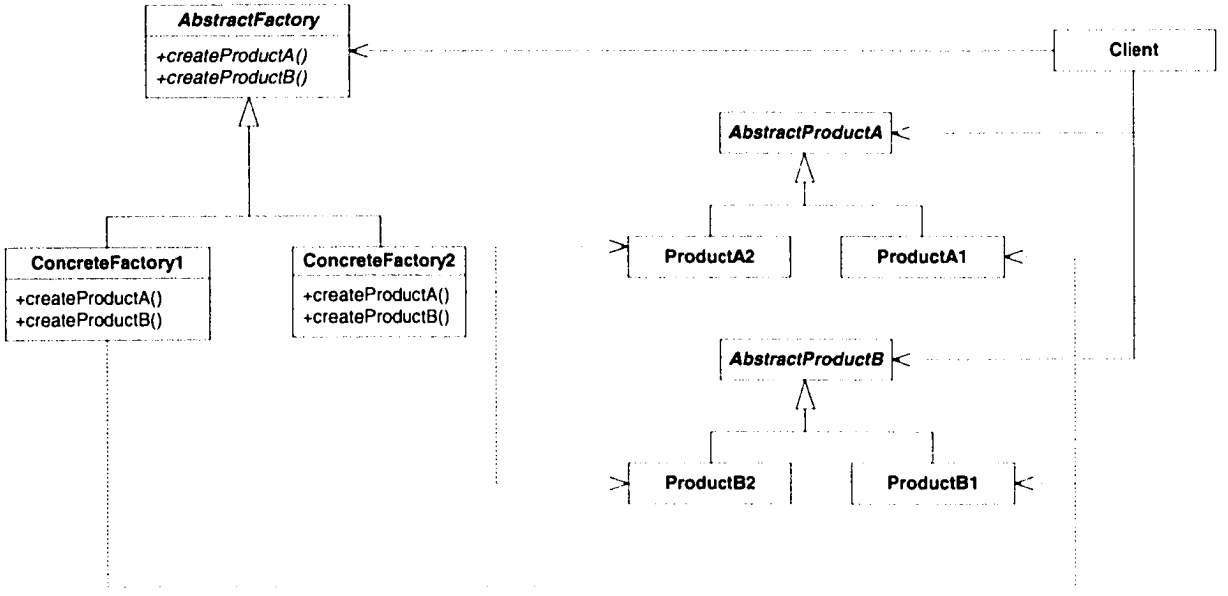

Figure 30 - *Abstract Factory* Structure [26]

The `AbstractFactory` class is then subclassed by implementation dependent classes (e.g. in Figure 30 – `ConcreteFactory1` and `ConcreteFactory2`, in Figure 34 – `PMWidgetFactory` and `MotifWidgetFactory`) which are responsible for the creation of each family set.

4.2.2 Applicability

As mentioned in [26] the *Abstract Factory* pattern is applicable in the following situations:

- A client should be abstracted from how its products are created, and implemented.
- A client must make use of multiple families of products.
- A family of related product objects is designed to be used together, and you need to enforce this constraint.
- The developer wants to provide a class library of products, and the goal is to completely abstract away the implementations of the classes that the client must use in this library.

The *Abstract Factory* pattern works in similar ways to the *Factory Method* pattern, in that it abstracts the creation of product classes away from the classes that make use of these products. The application of the *Abstract Factory* pattern is a little more demanding as there are new classes that must be created; however it does provide some ability to easily extend the system in the future that the *Factory Method* pattern does not.

4.2.3 Discussion of Pattern Candidate Structures

To determine candidate structures for the *Abstract Factory* pattern we first determine locations where the *Factory Method* pattern is applicable by identifying the candidate structures presented in section 4.1.3.1. After identifying the candidate structure for the *Factory Method* certain conditions are checked to determine whether the use of the *Abstract Factory* over the *Factory Method* is warranted. The remaining part of this section will discuss different situations in which the *Abstract Factory* pattern should be chosen over the *Factory Method* pattern due mainly to its ease of extendibility.

4.2.3.1 Package Case

The first situation to consider is whether the classes created inside the alternative combined fragment are from the same package. For instance in Figure 28, if `ConcreteProductA` and `ConcreteProductB` are found in the same package, as well as their first common generalization[2]; this may indicate that the *Abstract Factory* pattern is a logical choice. The factory can be placed within the package as to abstract the conditional creation as well as the true nature of the implementation of the various specific implementations from clients. The clients will only be required to have the interface to be used available, allowing the package to be altered without affecting the clients. If the client is from the same package, the *Abstract Factory* does not aid in masking the existence of the various implementation classes; therefore the *Factory Method* may be better suited.

The use of a factory to abstract the logic used to determine which implementation class to instantiate may not always be possible as the information used in the logic may reside within the client. The information being referred to here is the information used in the logical expression of the interaction constraint which determines the object created. There are means by which such information can be transferred to the factory instance if the information needed to make the decision is minimal and generic[3]. As an example, suppose that a package was developed to abstract the communication between a client and various servers and the choice on which class to instantiate depends solely on the version of the software found on the server. Figure 31 illustrates the pattern candidate structure for this example. Note that the `getVersion` call to the server as seen in the diagram is a simplification of what would be required from the client to determine the server's version. In reality this would require opening a new channel of communication to the server, creating a new packet with the request, receiving the packet from the server

---

[2] The 'first common generalization' is the most specific interface or ancestor class that two classes have in common.

[3] The meaning of 'generic' in this case is if the same condition applies to all clients using this library.

and parsing to retrieve the desired information. It is this form of low-level communication details the package is designed to abstract away from the client.

The use of the *Abstract Factory*, as mentioned previously, is used to abstract some of the creational details from the `Client` class.

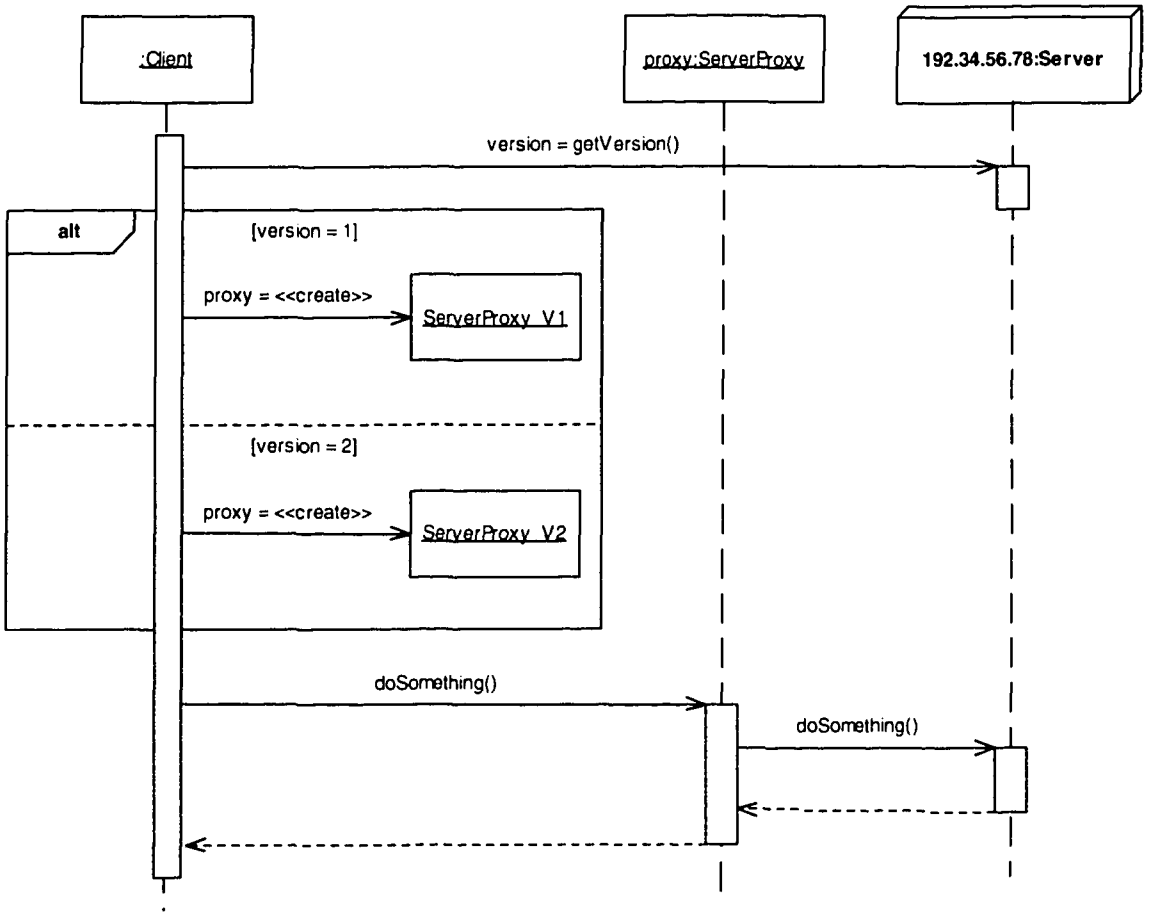

Figure 31 - Server Communication Example Sequence Diagram

In this instance it would be possible for the client to inform the factory instance as to which server they intend to connect, the factory can probe the server for the version of the software and return the proper object for the client to communicate with the server. The static structure for this application of the *Abstract Factory* is illustrated in Figure 32, followed by a sequence diagram where the pattern has been applied.

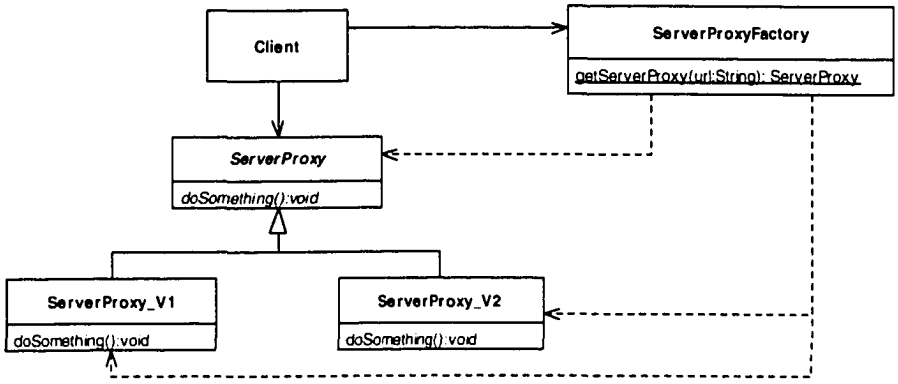

Figure 32 - *Abstract Factory* Applied to Server Communication

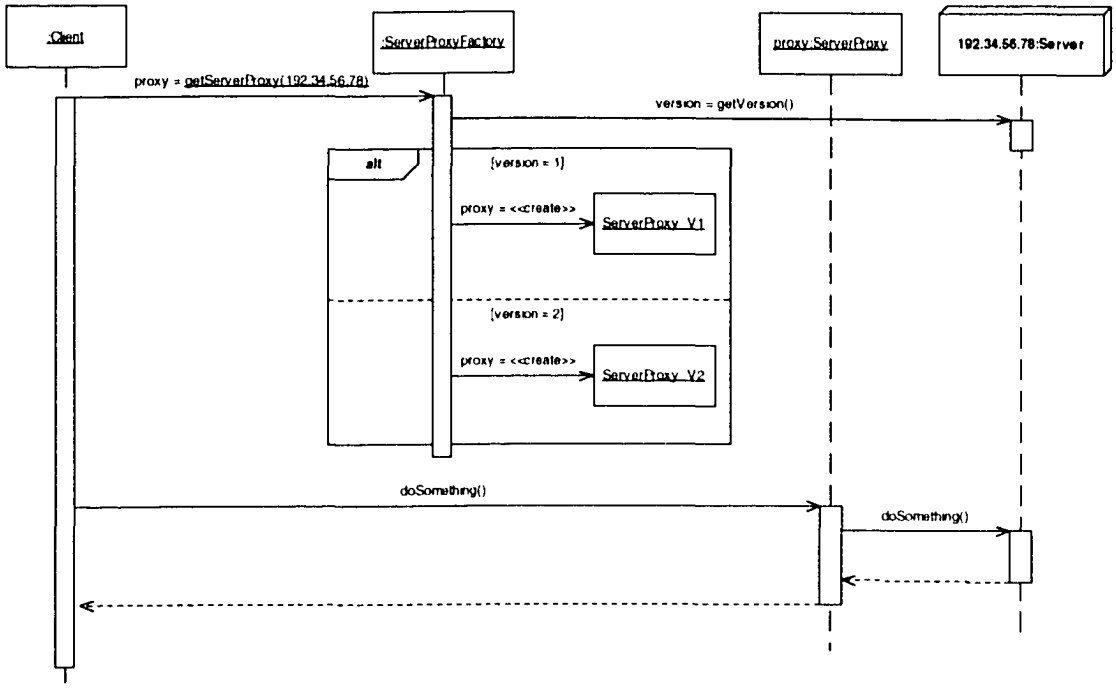

Figure 33 - Server Communication Example Sequence Diagram Using *Abstract Factory* Pattern

As can be seen in Figure 33 the factory abstracts away from the client the logic for determining which version of the `ServerProxy` would be needed. The condition in this case is considered generic as it will be the same for all clients of `ServerProxy`. The information as to which concrete class to create can be obtained in a variety of ways: at

runtime by the factory instance, can be provided by the client, or obtained through configuration files.

4.2.3.2 Several Clients

The next case to consider is when there are several clients creating the same group of objects. Suppose that several clients have to perform similar logic to create the product instance required, it would be sensible to abstract such condition into a factory; therefore if the logic changes or more classes are required this reduces the impact on the existing code. This abstraction for the client can be achieved by creating a factory class that contains the logic to determine which product class to create. Another means of achieving this is to make use of the *Decorator* pattern, but this alternative will not be discussed further here.

4.2.3.3 Several Families of Product Classes

The final case to consider is where several families of classes are being constructed within the alternative statements. This more closely resembles the application of the *Abstract Factory* shown in Figure 30. Figure 35 illustrates the dynamic behaviour of the candidate structure making use of the structures defined in Figure 34.

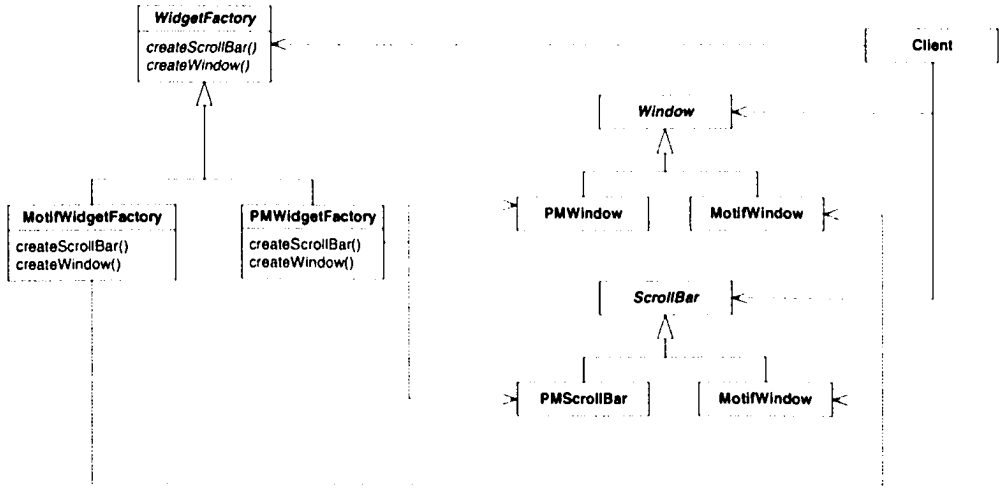

Figure 34 - *Abstract Factory* Structural Implementation [26]

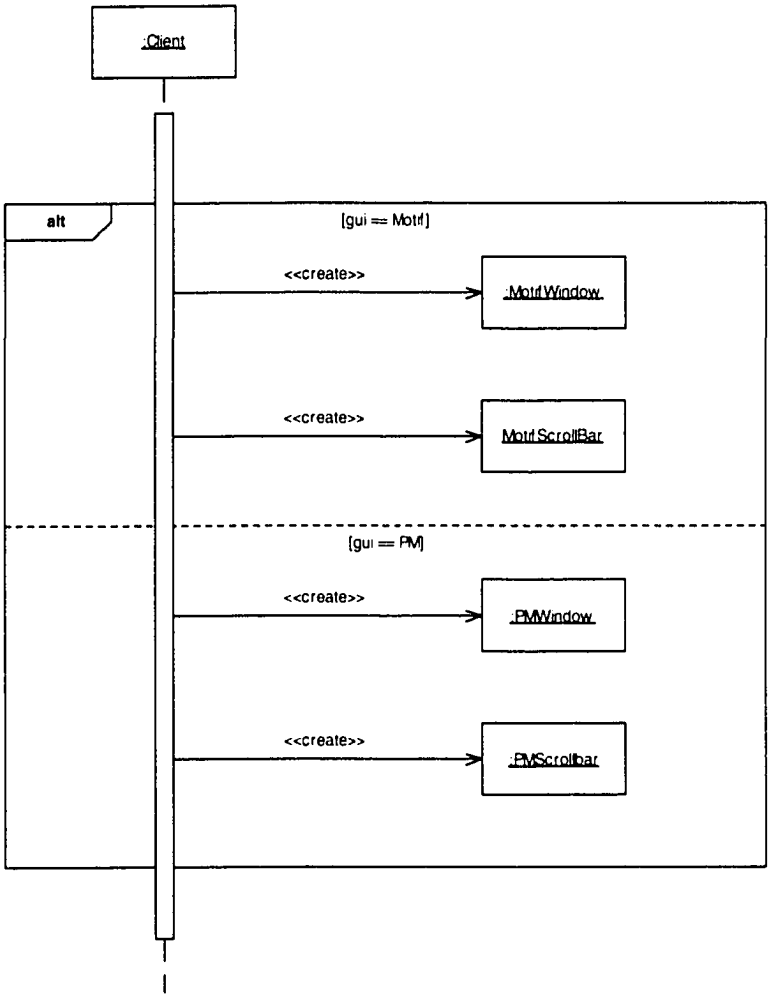

Figure 35 - *Abstract Factory* Opportunity with Multiple Families

In this case it would be wiser to implement the *Abstract Factory* pattern rather than having two or more factory methods to maintain in each client. Another manner of handling this situation is through the use of attributes and the *Factory Method* pattern; however this solution is less elegant and will not be discussed further.

4.3 Identification of Candidate Structures for Factory Method/Abstract factory patterns

This section of the report presents the meta-model and detection rules used to determine potential locations for the use of the *Factory Method* or *Abstract Factory* pattern. We begin with the definition of the meta-model describing the information required from the UML diagrams to properly identify the candidate structures. A decision tree is generated to identify particular candidate structures in the system. The decision tree will consist of three types of nodes.

- *Automated Analysis Node:* It represents some condition that can be checked again the UML model of the system. The condition is specified by the use of OCL rules, as they provide an unambiguous and logical means of describing the conditions.

- *User Interaction Node:* It represents a query to the user to obtain information that cannot be obtained from the UML diagrams.

- *Pattern Suggestions:* the leaf nodes of the decision tree which indicate that a particular pattern is suggested for the structure identified by the decision tree.

Each node can generate information that is required by subsequent nodes. This information is made available through the use of *in/out* parameters. The return value of the nodes guides the navigation through the tree and if there is no navigation available for the result this indicates that no pattern suggestion can be provided.

The OCL [40] rules developed are to be as precise as possible in that they make use of all relevant information available in the UML diagrams. At the same time the rules should be as minimal as possible, in that they should only perform the minimal analysis required to derive an accurate and complete solution for detecting candidate structures. The rules developed for this report do not identify the second alternative candidate structure

presented in section 4.1.3.3. These rules will be refined as to take into consideration this structure at a later stage.

This report focuses on the logic and conditions required for identifying the candidate structures. At a later stage the OCL rules will have to be implemented as algorithms and this may again result into changes in the meta-model. But at this stage recall that our main objective is to identify whether detection rules can be devised for common design patterns.

4.3.1 Meta-Model

In order to perform analysis on the UML documents generated for the system, a meta-model of the UML diagrams was created (Figure 36). This meta-model only contains the information required for determining locations for the *Factory Method* and *Abstract Factory* (presented in section 5). For a complete meta-model that presents all the information used for all the candidate structures presented in this thesis, refer to Appendix A. The meta-model is inspired from the standard UML 2.0 meta-model [41] but is adapted to our needs.

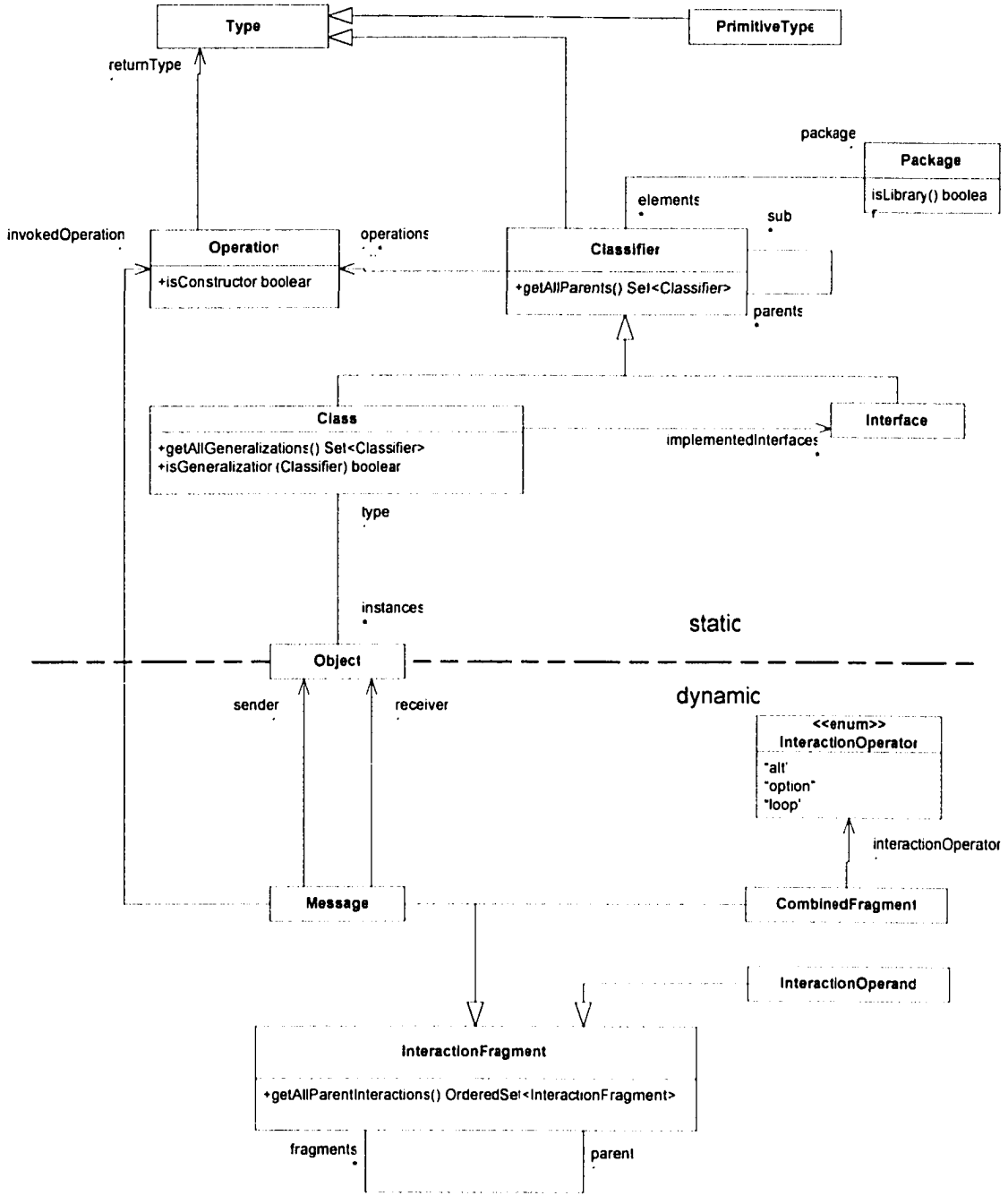

Figure 36 - Meta-Model for *Abstract Factory* and *Factory Method* Patterns

To aid in understanding how this model is instantiated an example sequence diagram with the corresponding instantiation is presented in Appendix B. A brief description of some of the less intuitive elements is detailed below:

- *InteractionFragment:* This is the basic element of the sequence diagram; can have sub-interactions and a parent interaction.

- *InteractionFragment::getAllParentInteractions():* Returns all the parent interactions or the call stack leading to this interaction.

```
post:
result = self.parents->union(self.parents->collect(p|
p.getAllParentInteractions())
```

- *CombinedFragment:* It defines a set of interaction fragments.

```
inv:
self.interactionOperator = InteractionOperator::alt implies
self.fragments->forAll(i:InteractionFragment|
i.oclIsKindOf(InteractionOperand))
```

- *InteractionOperand:* It defines a set of interaction fragments whose execution is guarded by a logical expression.

```
inv:
self.parent.oclIsKindOf(CombinedFragment)
```

- *Classifer::getAllParents():* Returns the inheritance hierarchy for this classifier.

```
post:
result = self.parents->union(self.parents-
>collect(p|p.getAllParents()))
```

- *Class::allGeneralizations():* Returns all the generalizations for this class.

```
post:
result = self.implementedInterfaces->
union(self.implementedInterfaces.allParents())->
union(self.parents)->union(self.parents->
collect(p|p.getAllGeneralizations()))
```

- *Class::isGeneralization(classifier:Classifier):* Test if the classifier is a generalization of the class.

```
post:
result = self.allGeneralizations()->includes(classifier)
```

4.3.2 The Factory Method and Abstract Factory Decision Tree

One example tree for the *Factory Method* and *Abstract Factory* patterns is presented in Figure 37. This decision tree is developed to iterate over the alternative combined

fragments in the system. The combined fragments are supplied to the decision tree through the `combFrag` parameter. The tree is designed to perform fast and simple tests at the beginning that will eliminate many of the alternative combined fragments very rapidly.

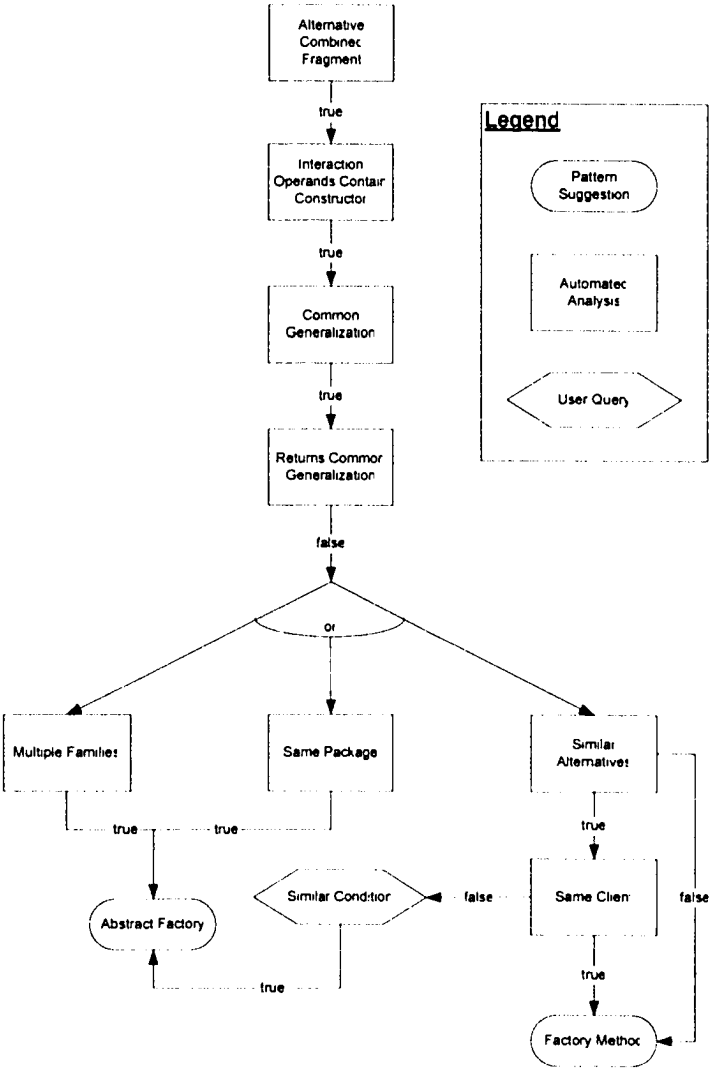

Figure 37 - *Factory Method* and *Abstract Factory* Decision Tree

This tree should be executed with every alternative combined fragment in the system sequence diagrams. The corresponding OCL rules that describe the analysis for each of the automated node is found in Appendix C. A textual description of the analysis performed at each node is given below, along with the parameters in input and output.

- *Alternative Combined Fragment:* It checks whether the combined fragment is an alternative combined fragment.
    - *in combFrag:CombinedFragment* - the combined fragment being tested.

- *Interaction Operands Contain Constructors:* It checks whether all the interaction operands within the combined fragment contain a constructor.
    - *in combFrag:CombinedFragment* - the combined fragment being tested.

- *Common Generalization:* It checks whether the classes being instantiated within the combined fragment have at least one higher-level generalization which is common; this is an indication that the classes are part of a generalization tree. A check should be performed to ensure that the common generalization is not trivial (i.e. the common ancestor is `java.lang.Object` or `java.lang.Comparable`).
    - *in combFrag:CombinedFragment* - the combined fragment being tested.
    - *in trivialGeneralizations:Set<Classifier>* - the set of classifiers that would be considered to be a trivial generalization.
    - *out generalizations:Set<Classifier>* - the set of generalizations that are shared amongst the classes created in the interaction operands.

- *Returns Common Generalization:* It checks if the method in which this combined fragment is found returns one of the generalizations. This indicates the *Factory Method* may already be applied.
    - *in combFrag:CombinedFragment* - the combined fragment being tested.
    - *in generalizations:Set<Classifier>* - the set of generalizations that are shared amongst the classes created in the interaction operands.

- *Multiple Families:* It checks whether there are multiple constructors found inside each interaction operand and multiple families of classes being created within the combined fragment. In the case of multiple generalization trees, the *Abstract Factory* pattern is a better choice then the *Factory Method* pattern.
    - *in combFrag:CombinedFragment* - the combined fragment being tested.

- *Same Package:* It checks whether all the classes created within the combined fragment, which share a common ancestry found in the same package.
    - *in combFrag:CombinedFragment* - the combined fragment being tested.
    - *in generalizations:Set<Classifier>* - the set of classifiers that would be considered to be a trivial generalization.

- *Similar Alternatives:* Given a combined fragment under consideration, for all other combined fragments in the UML sequence diagrams, checks whether the classes created within an interaction operand in one block correspond to those of one interaction operand in the other block to a given degree. The goal here is to locate similar combined fragments in the system.
    - *in combFrag:CombinedFragment* - the combined fragment being tested.
    - *out similarCombFrags:Set<CombinedFragments>* - the set of combined fragments that are deemed similar to combFrag.

- *Same Client:* It checks whether the client is the same for both combined fragments under consideration?
    - *in combFrag:CombinedFragment* - the combined fragment being tested.

- *in similarCombFrags:Set<CombinedFragments>* - the set of combined fragments that are deemed similar to combFrag in the previous analysis.

- *Similar Conditions:* It inquires from the user if the conditions used in the alternatives blocks are similar. A user interaction is required.

- *Abstract Factory:* The decision tree suggests that the structure(s) discovered may be a candidate for the *Abstract Factory* pattern.

- *Factory Method:* The decision tree suggests that the structure(s) discovered may be a candidate for the *Factory Method* pattern.

4.3.3 Scalability

When considering the scalability of this decision tree or other decision trees developed in this research, all begin with filtering the UML model for a particular type of UML element. The filtering for a particular type of UML element will scale linearly with the size of the UML model being analyzed; however the algorithms should then be assessed based on the scaling in relation to this particular UML type as this is what is variable between all the algorithms.

In the algorithm presented here the UML element of interest is the combined fragment. The worst case time is $O(N^2)$ (N represents the combined fragments in the UML model) as in the worst case scenario would have all combined fragments would be instantiating objects, therefore forcing that all combined fragments be crossed compared in the 'Similar Alternatives' node. In reality the performance of this algorithm will be greater as not all combined fragments will meet the initial conditions (Alternative Combined

Fragment, Interaction Operands Contains Constructor, Common Generalization, Returns Common Generalization) at the beginning of the decision tree.

The scalability of this algorithm can also be improved by filtering at the beginning of the decision tree all the combined fragments for those that are alternative combined fragments that contain object instantiations in each of the alternatives. The decision tree would then make use of this reduced subset of combined fragment to perform its analysis. This would not affect the worst case time complexity (not a realistic situation), however would impact the average execution time.

4.3.4 Discussion

The issue that should be highlighted with this decision tree is that this approach may identify correct implementations of the patterns in the system. The mechanism to ensure that most of these implementations are not detected is the check on the return value from the method. If the *Abstract Factory* or *Factory Method* design patterns were used the return value from the method in which the combined fragment resides would return a common generalization of the types instantiated within the combined fragment. This test is crucial as to reduce the number of false positives generated by the tool.

5 OBSERVER PATTERN

5.1 Description

There are many cases when cooperating objects need to maintain consistency. The *Observer* pattern is used to achieve this consistency without incurring a high coupling between the classes and thus reducing the reusability of the system. The *Observer* pattern describes how to establish a relationship between two objects, where one object (observer) needs to be notified when changes of state occur in the other object (subject). The general structure of the *Observer* pattern is given in Figure 38.

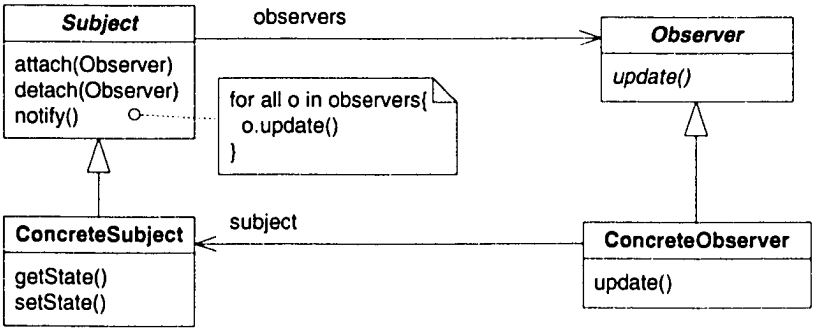

Figure 38 - *Observer* Pattern Structure [26]

5.2 Applicability

As defined in [26] the *Observer* pattern is applicable in a variety of situations:

- When a state change of one object requires that others be updated as well. Note that it is not really applicable when there is only a single observer of a single type, in this case the *Observer* pattern would be excessive.

- When one object is responsible for notifying other objects without assuming what objects will require the update. This is simply to reduce the coupling between objects.

5.3 Discussion of Pattern Candidate Structures

Identifying locations in the system where the *Observer* pattern could be applied requires identifying the locations in the system where a similar behaviour is achieved without the good practice of the design pattern; such a situation is modeled in Figure 39.

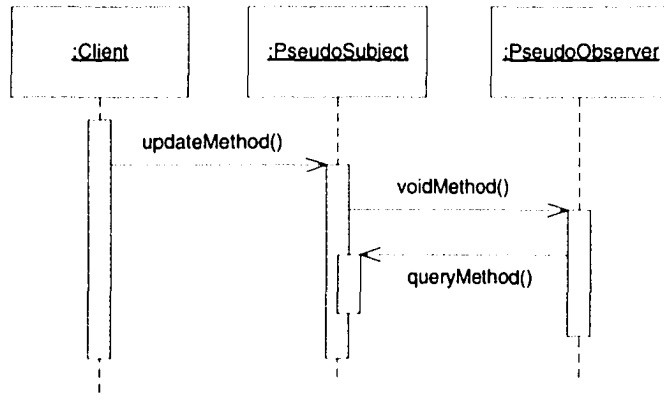

Figure 39 - Candidate Structure for *Observer* Pattern

The simplest way of identifying this candidate structure is to locate the `voidMethod` and `queryMethod` invocations. The approach taken begins by examining a simple message (referred to hereon as the 'original message') sent to a destination (in the example given this would be the `PseudoObserver`). The original message should be followed by an invocation of a query[4] method on the original sender (in the example given this would be the `PseudoSubject`) originating from destination object. In summary, as mentioned in the Applicability section, this indicates that one object is updated and others are notified of this update. The original message should have been caused by a change in the state of

[4] A query method is one that does not alter the internal variables or state of the object it is executed on. They are often referred to as getter methods. [3]

the `PseudoSubject` object; therefore it should be preceded by an update method[5] invocation to this object.

The level[6] of the update method may not be known in relation to the original message. The reason for this is that a user may have followed some good practices by encompassing the notification of the `PseudoObserver` in a private method. This is shown in Figure 40 with the `notify` method.

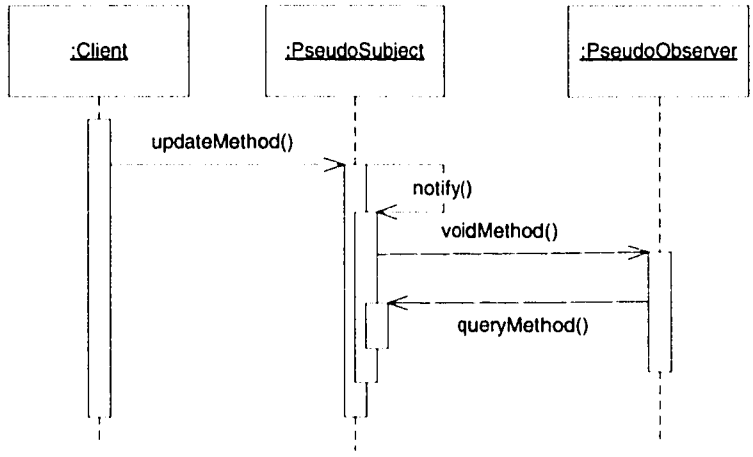

Figure 40 - Alternate Candidate Structure for *Observer* Pattern

This is close to the *Observer* pattern; however this structure does limit the system to a single observer for the `PseudoSubject`. Also, in the pattern, the `notify` method should be declared in a super class or interface that concrete subject inherits from.

These candidate structures follow the main principle of the *Observer* pattern: objects are updated once a state change is performed on the subject object. Figure 41 presents a sample sequence diagram for the *Observer* pattern built on the static structure of the pattern presented in Figure 38. The structures shown in Figure 39 and Figure 40 are not technically wrong, but they do limit the number of observers, and potentially the set of

---

[5] An update method is one that alters the internal variables or state of the object it is executed on. They are also often referred to as setter methods. [3]

[6] Level refers to the level in reference to the parent/subInterations relationships.

classes that may observe, thus limiting the overall extensibility of the system. Therefore the developer should be made aware of situations were only some of the concepts of the pattern were applied.

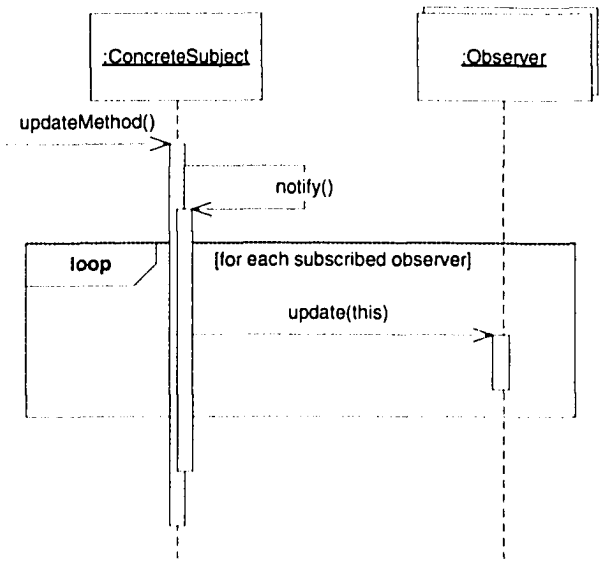

Figure 41 - Observer Interaction

There are key concepts to the *Observer* pattern, and once the `voidMethod` and `queryMethod` are identified the approach taken is to ensure that all the key concepts of the pattern have been applied. It is true that there are situations were not all the concepts should be applied (e.g., if there is only a single observer the loop structure would not be required); however these situations are rare. If all the concepts are applied then it assumed that the structure is a proper implementation of the pattern. By using this approach alternative candidate structure such as the one presented in Figure 40 will be identified. There are three key concepts to be checked:

1. If the *Observer* pattern is properly implemented there must be a self-call to a method inherited from the super class (i.e. `notify`). It is also important to look for methods defined in interfaces, especially when the programming language only supports single inheritance (e.g., Java, C#). Indeed, in case of single inheritance, `Subject` may be an interface defining the signature of `notify()`, and

the management of the observers (the implementation of `notify`) would be directly in the concrete subject, as suggested in [34]. This however leads to redundant code in all classes that implement this interface. However in this case there should still be a self-call with the method invoked being defined another classifier than the concrete subject.

2. Determine if the call to the observer(s) is placed inside a loop statement. If the *Observer* pattern was implemented there would potentially be many observers to update and this would definitely be implemented as some form of iteration over the list of subscribed observers. As mentioned earlier there are rare situations when a subject will only ever have a single observer. In this situation the loop structure will not be needed.

3. Ensure that the type used for the observers is an interface (or an abstract class). If instead the observer type is a (concrete) class this would limit the extensibility of the system, by limiting the types of classes that can be observers to those that inherit from the observer class. Again there are rare situations were only classes within a given hierarchy would be observers, therefore the observer methods would be placed at the root class of the hierarchy.

5.3.1 Identifying Query Methods

There are three ways to determine whether a given method is either an update or query method. They are presented below in the order that the tests would be performed.

1. Using stereotype to describe the nature of the operations of a class. There are standard stereotypes for operations as described in [2] that indicate if an operation is either a query operation (<<query>>) or an update operation (<<update>>). This stereotype information may be gathered from the class diagram, if it is provided by the developer.

2. Using the OCL contracts for the method. If the post-condition(s) of the method consists of a single statement describing the return value this indicates that the method is a query method. On the other hand, if the post-condition(s) contains other statements, this may indicate that the method is an update method. This is a general case as those additional statements may describe how output parameters are set instead of modifications to the system state, thus indicating a query. This is not as common, and is not further investigated for simplicity.

3. Using the signature of the method. This test is a rather weak test that relies on the conformance to standard practices in naming operations for the Java language [3]. However if this convention is used most query methods should be preceded by 'get' or 'is', while most update methods should be preceded by 'set'. There are cases when query or update methods will not have the preceding identifier even when the convention is followed. A good example of such a situation is a collection class (such as `java.util.Collection` [36]) where there are several update and query methods but none follow the convention mentioned.

The first two tests cannot be done on UML diagrams that are generated through static analysis of the code. Such diagrams will not contain the OCL contracts for the operations, nor will contain accurate stereotypes for the operations. If the naming convention is followed the signature can be used to deduce the nature of the operation. The UML standard [41] does provide an attribute, `isQuery`, on the `Operation` type that can be used to determine the nature of the operation. Note that the value of the `isQuery` attribute will be based on the three tests provided above, and if none of the tests can be adequately performed then the user must provide this information.

5.4 Identification of Candidate Structures for the Observer Pattern

As done for the *Factory Method* and *Abstract Factory* patterns, this section presents a stepwise approach for determining locations to apply the *Observer* pattern.

5.4.1 Meta-Model

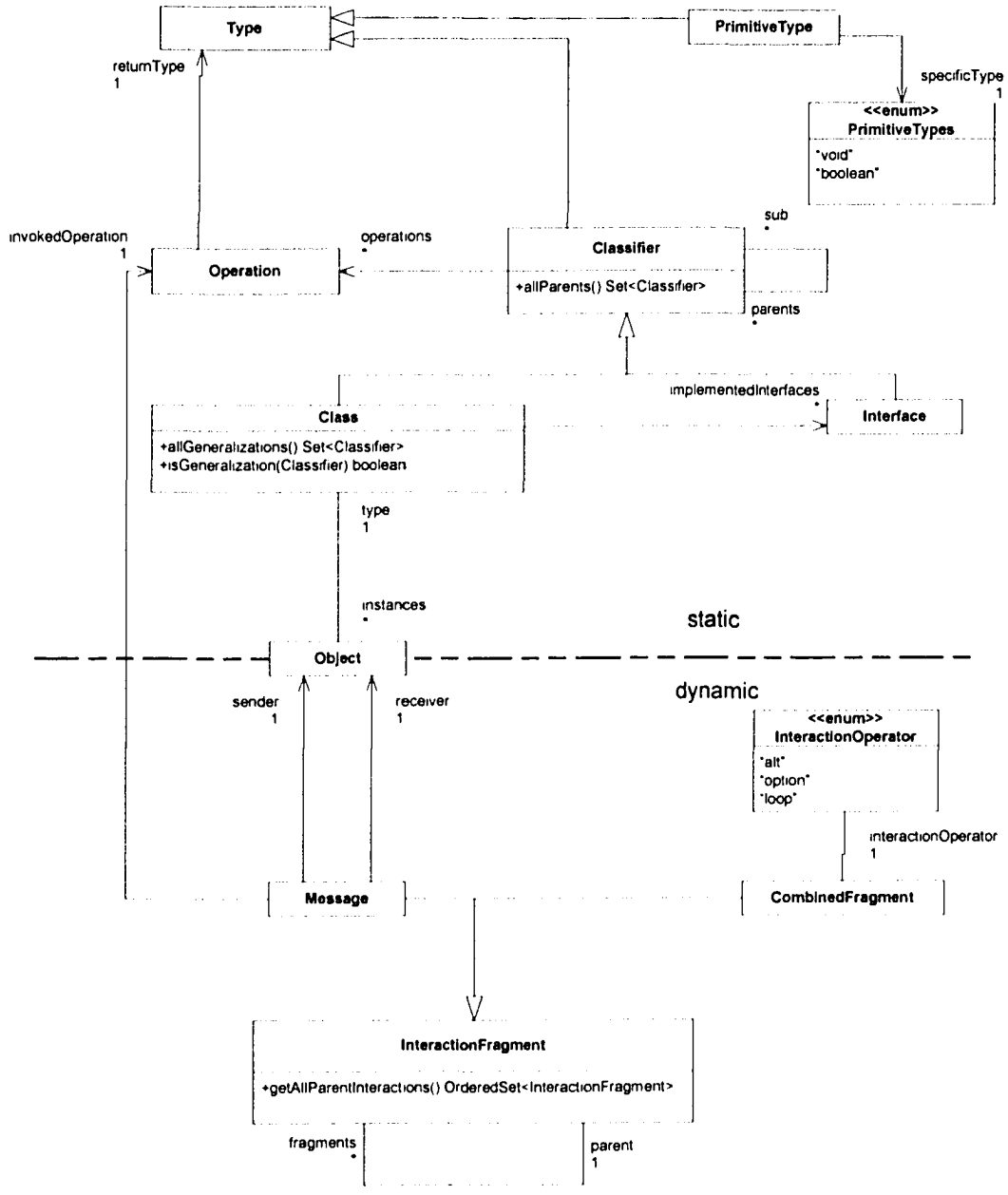

Figure 42 - Meta-Model for *Observer* Pattern

Figure 42 presents the meta- model for the *Observer* Pattern. This meta-model is presented to specify the information used to identify the candidate structures presented in section 5.3.

5.4.2 Observer Pattern Decision Tree

Figure 43 presents the decision tree for the pattern. The tree works by iterating over the messages found in the sequence diagrams of the system. The messages are supplied to the decision tree through the `observerInteraction` parameter. The OCL rules for each of the nodes are given in Appendix D.

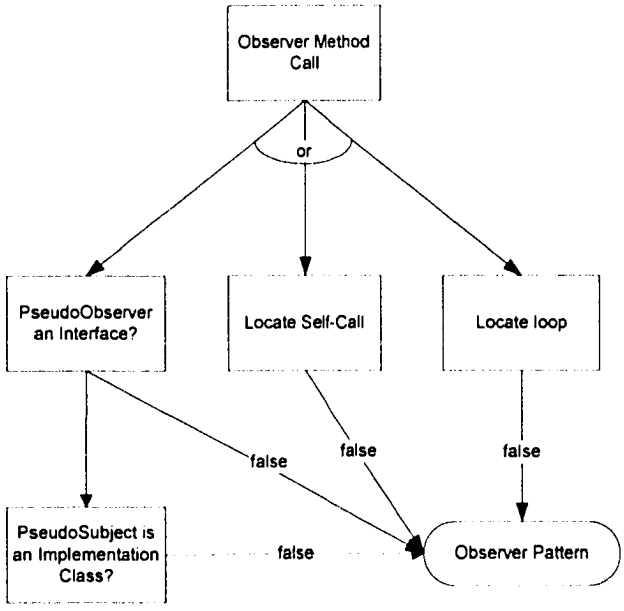

Figure 43 - *Observer* Pattern Decision Tree

The descriptions of the individual nodes in the tree are found below, along with their input and output parameters:

- *Observer Method Call*: Firstly checks whether the message (i.e.observerInteraction) is an update message without any return value. Secondly it checks if the message is followed by query method invocation back from the destination object of the first message to the sender of the first message. An example sequence of such message is illustrated in Figure 39 with the `voidMethod` and `queryMethod` calls.
    - *in observerInteraction:Message* - the message being investigated.

- *PseudoObserver is an Interface:* It checks whether the `PseudoObserver` class is an interface. This would allow for any class to observe the state of the subject, by ensuring that they implement the interface. If the `PseudoObserver` was a class type this would limit the observers of the subject to those classes within the inheritance hierarchy. There may be times when this is a valid approach, but more often the versatility of the interface is highly desirable.
    - *in observerInteraction:Message* - the message being investigated.
    - *out observerInterface:Interface* – the interface used for the observer type.

- *Locate Self-Call*: This checks whether the original call is within some self-invocation. An example of a self-call is the `notify` invocation found in Figure 41. The importance of the self-invocation is to reduce code duplication. Once the notification is placed inside a method, it is easy to add notifications on the class or other subclasses when needed.
    - *in observerInteraction:Message* - the message being investigated.
    - *out selfCall:Message* – the self invocation.

- *Locate Loop*: This checks whether the original call was located in a loop structure. If it was not found in a loop structure this limits the number of observers and thus the extensibility of the system.
    - *in observerInteraction:Message* - the message being investigated.

- *PseudoSubject is an Implementation Class:* It checks whether the `PseudoSubject` class is a subclass (of an abstract class) or implements some interface. One requisite for the interface/super class is that it has operations for adding or removing observers. The check for this is rather simplistic; simply check for operations (two or more) with a parameter whose type is the

`PseudoObserver` type (output of the *PseudoObserver is an Interface* node). The reason for the `PseudoSubject` to have a higher level of abstraction for the subject is that other classes may play the role of the subject. Using an interface provides the most versatility to the system, while using an abstract class provides the most reuse in the code. The trade-offs between the two are straightforward, but designers should make use of one to these mechanisms in the case of future subjects being introduced in the system.

  - *in observerInteraction:Message* - the alternative combined fragment being tested.
  - *in selfCall:Message* – the self invocation
  - *in observerInterface:Interface* – the interface used for the observer type.

- *Observer Pattern:* indicates to the user that this structure in the system may be an ideal place to make use of the *Observer* pattern.

5.4.3 Scalability

This algorithm will scale linearly, $O(N)$ with respect to the number of messages found in the UML model. This is quite significant as the number of messages can grow quite high as they are a common UML type in sequence diagrams.

5.4.4 Discussion

This decision tree will return structures that are very closely related to the proper implementation of the pattern but missing certain key features of the pattern that allow for reuse and extensibility. As can be clearly seen from the decision tree, the approach begins by identifying methods similar to the `voidMethod` and `queryMethod` as seen on Figure 39. The remaining analysis checks for specific traits of the *Observer* pattern that permits the extensibility of the system. If the checks are all successful then the structure identified is most likely a proper implementation of the *Observer* pattern: Figure 43

shows that such a situation is not reported. The discussion as to why each check is important follows from the description of the analysis performed at that node in the decision tree.

The problem with this approach is that there are situations where some of the concepts of the *Observer* pattern should/could be ignored. A clear example of this is when there is ever only a single observer; in such a situation the loop structure is not needed. These situations should arise fairly infrequently, thus should not severely impact on the accuracy of the approach. If it is found that this does impact on the accuracy then further research will be required to modify this decision tree.

6 DECORATOR PATTERN

6.1 Description

The *Decorator* pattern is used to extend the functionality of an object at run time. The *Decorator* pattern allows for this extension without the need to alter the original implementation and without the use of inheritance. The functionality is added by wrapping the original object in a `Decorator` object that provides the desired functionality. The structure of the *Decorator* pattern is given in Figure 44.

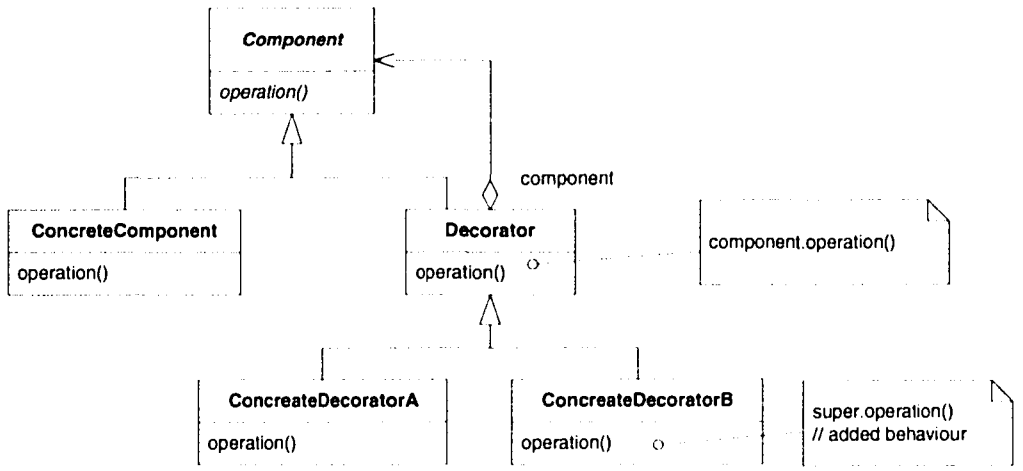

Figure 44 - *Decorator* Structure [26]

6.2 Applicability

The *Decorator* pattern applies in a variety of different situations [26]:

- Can be used to add responsibility to objects at run time without affecting the original object.
- Can be used when responsibilities must be removed from objects at run-time.

- To avoid extensive use of subclassing when new functionality is added. When independent functionality is added to classes of a hierarchy this can result in an explosion of subclasses to support the various possible combinations.

6.3 Discussion of Pattern Candidate Structures

The key to the candidate structure for this pattern is given in the third point of applicability. When new functionality is to be introduced in a system, the accepted approach is to use inheritance. As an example, let us assume that there is a class responsible for logging messages to the console, `ConsoleLogger`. This message logging component is to be extended and provide XML log messages to the console. The standard approach is to subclass the `ConsoleLogger` to create the `XMLConsoleLogger`. This is depicted in Figure 45.

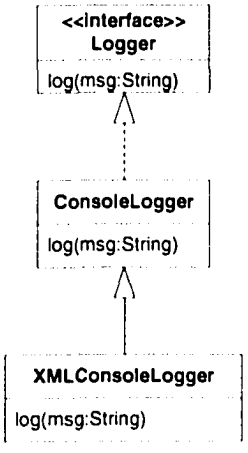

Figure 45 - XML Extension to Console Logger

This leads to an elegant design if the number of extensions is limited and the added functionality is not independent from one another. In this example, the functionality is independent. `ConsoleLogger` provides a means to write logs to the console, while `XMLConsoleLogger` simply alters the log messages. The functionality of writing the log to a physical device is independent from the formatting of the log message. In this example the hierarchy remained simple due to the small number of

extensions/functionality introduced. In a case where there are several various types of extensions that are required and for the most part these extensions are independent, the result can be a large hierarchy where all possible combinations must be exercised. This is demonstrated in Figure 46.

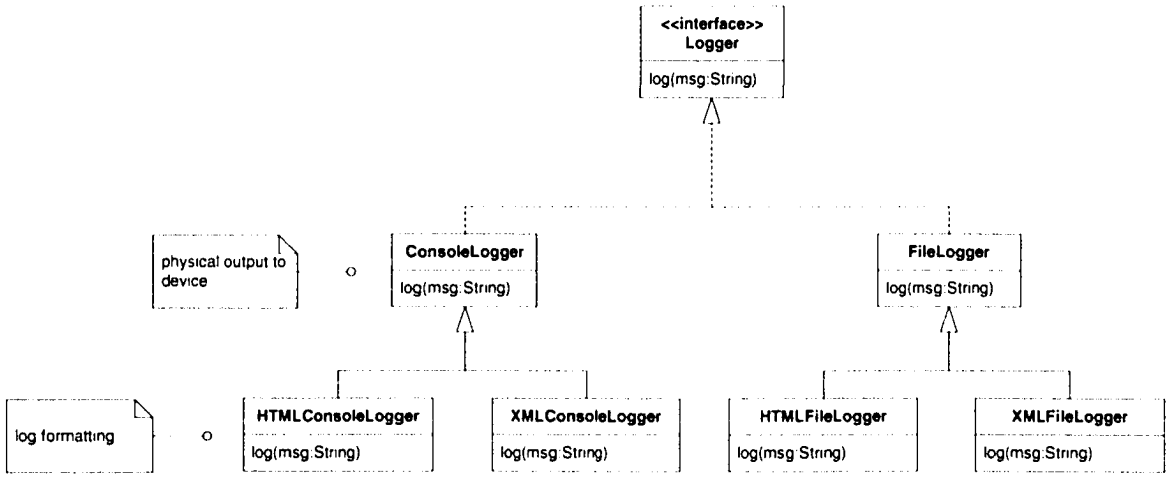

Figure 46 - Complicated Hierarchy Resulting from Subclass Extensions

To discover a similar hierarchy in the system is rather straight forward: search for a large hierarchy where subclasses offer minimal extensions to the interface offered by the parent. If a subclass conforms very closely to the interface exposed by its parent, this suggests that the purpose of the subclass is to extend the functionality of the parent. If a whole hierarchy shows such conformance to a single interface this would suggest that the *Decorator* pattern may be applicable. The application of the *Decorator* pattern to the hierarchy from Figure 46 can reduce the number of classes and allows for easier extension of the system. This is illustrated in Figure 47.

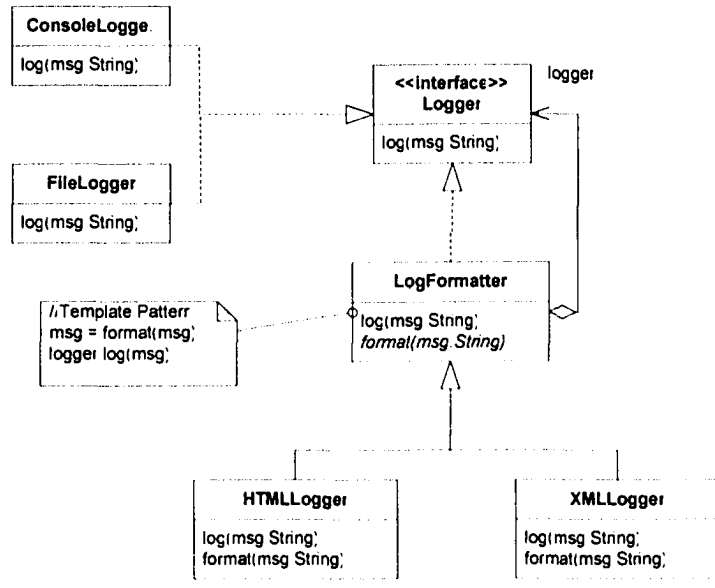

Figure 47 - Application of *Decorator* Pattern

One of the difficulties faced when identifying this candidate structure is to determine what is to be considered a close conformance to an interface. It seems logical use a ratio of public methods defined in the parent versus the new methods introduced in the subclass. To determine the proper ratio may take some research into existing systems containing the type of hierarchies being mentioned. Like the detection rules developed in this research this ratio will be constantly adjusted based on empirical testing.

Another difficulty with this approach is to avoid returning implementations of the *Template Method* pattern as possible candidate structures. The *Template Method* pattern makes use of inheritance to extend functionality of the system. An example of the *Template Method* pattern can be found in Figure 47, it is used for the `LoggerDecorator` and its descendants. The overridden methods in the *Template Method* pattern should be declared as protected, this permits overriding by descendants and limits accessibility to clients; therefore the public interfaces of the descendants do not deviate widely from the interface of the ancestors thus making implementations of the *Template Method* susceptible to be returned as candidate structures. The *Template Method* however can be over used. If applied to a large hierarchy where the extended functionality is independent

the implementation of the *Template Method* pattern would not reduce the size of the inheritance hierarchy. Figure 48 illustrates the application of the *Template Method* pattern to the hierarchy presented in Figure 46. Therefore whether the hierarchy is a result of the *Template Method* being applied or not, the deciding factor should be one of size and complexity. If the *Template Method* is improperly used a large hierarchy can arise, the driving cause is the independent functionality and the need for all possible permutations. Therefore larger hierarchies are still suspect for the possible introduction of the *Decorator* pattern and some research will have to be performed to properly determine the size of hierarchies to be considered as candidate structures.

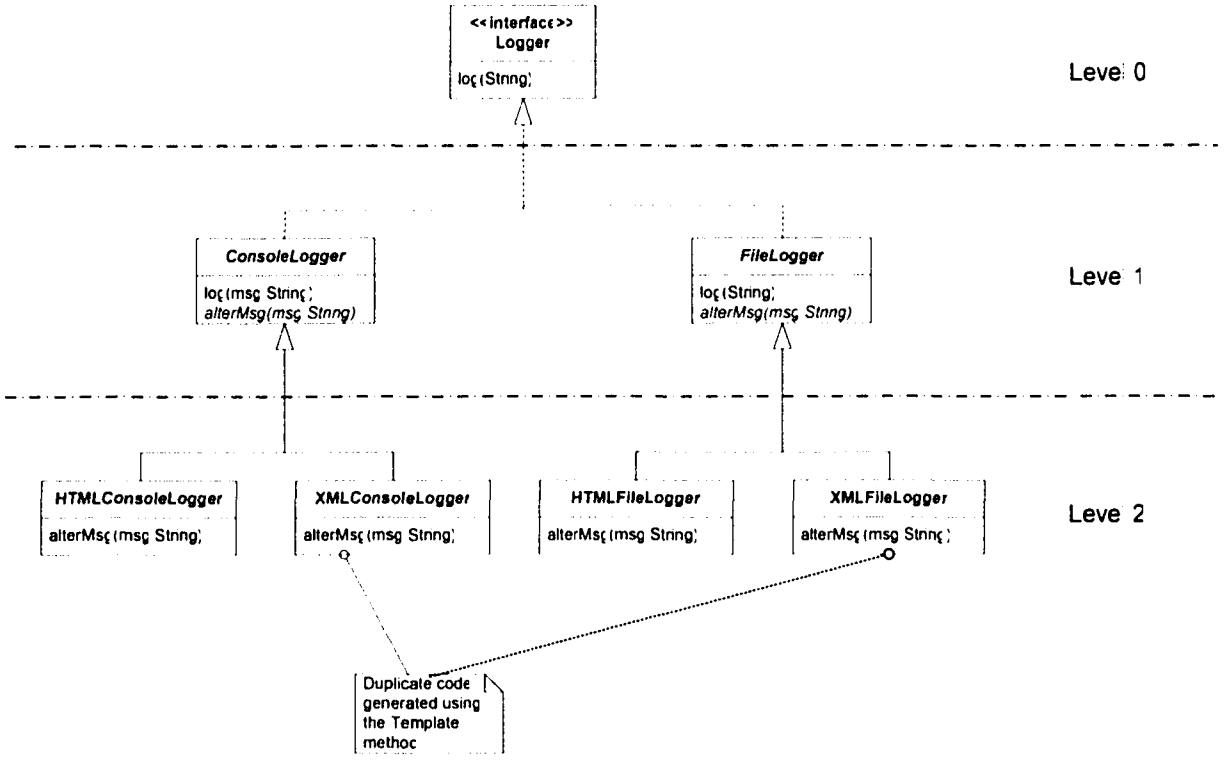

Figure 48 - Template Pattern Implementation

The independent functionality is an important factor in identifying this candidate structure. An important area would be the possible automation of determining whether the functionality is independent. This may be possible by examining the structure of the hierarchy. Note that if the functionality is independent and that all the possible

combinations were exercised it would result in every class at a given level in the hierarchy having the same number of children (this property will be referred to as level completeness). This is an easy quality to test for and is illustrated with the hierarchies in Figure 46 and Figure 48. However this may not always be the case, there may be situations where not all permutations are necessary or other factors that can lead to level incompleteness; therefore ultimately the decision of whether the functionality can be considered independent must fall on the user. An automated check can be performed before querying the user. The automated check will analyze the hierarchy checking for level completeness. If the automated check fails then the user is queried to determine if the functionality is independent; however if the test succeeds then assume that the functionality is independent. If the automated test does succeed while the functionality is not independent the user can simply disregard the suggestions proposed by the tool. This automation will reduce the need to interact with the user, but at the same token may decrease the accuracy of the suggestions.

6.4 Identification of Candidate Structures for the Decorator Pattern

6.4.1 Meta-Model

The meta-model required to explore this candidate structure is a simplified version of that presented in section 4.3.1 and is given in Figure 49.

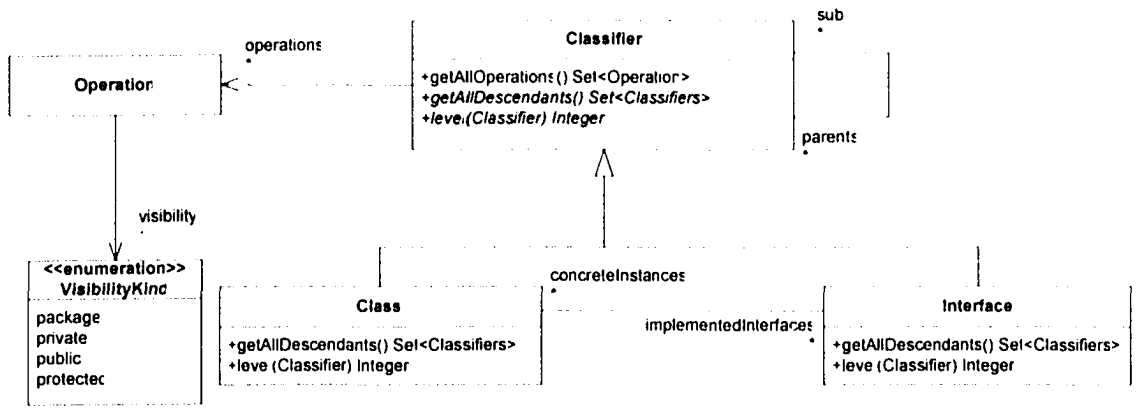

Figure 49 - Meta-Model for *Decorator* Pattern

A new method is added to the `Classifier` to provide the inheritance level of the classifier within a given hierarchy. The concept of level is illustrated on Figure 48, relative to the `Logger` interface. If the classifier is not a descendant of the root classifier supplied the method returns a value of -1. Note that if there are two different paths possible within the hierarchy from the root to the classifier, the smaller path is returned. The rest of the methods are described through OCL below.

- *Classifier::allOperations():* Returns all the operations supported by this classifier, even those defined on parents.

```
post:
result = self.operations->union(self.parents-
>collect(p|p.getAllOperations()))->asSet()
```

- *Class::getAllDescendants():* Returns all the classes found in the inheritance hierarchy.

```
post:
result = self.sub->union(self.sub->collect(s |
s.getAllDescendants()))
```

- *Interface::getAllDescendants():* Returns all the classifiers to whom this classifier can be considered a generalization.

```
post:
result = self.sub->union(self.concreteInstances)-
>union(self.sub->collect(p|p.getAllDescendants())-
>union(self.concreteInstances-
>collect(c|c.getAllDescendants()))->asSet()
```

6.4.2 Decorator Pattern Decision Tree

The decision tree for this pattern is simple and given in Figure 50.

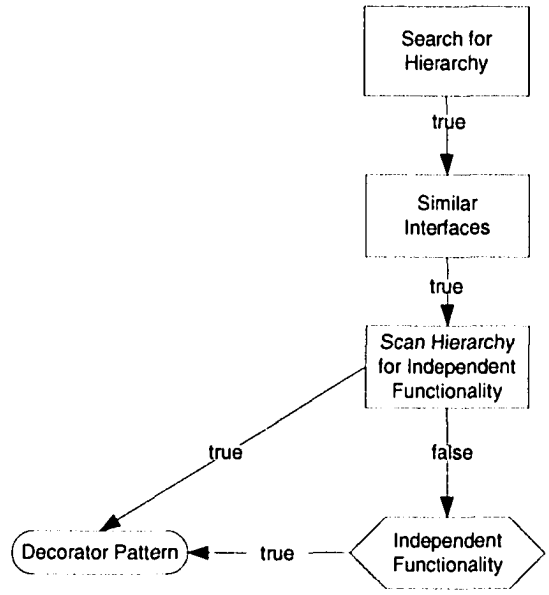

Figure 50 - Decision Tree for *Decorator* Pattern

A brief description of the analysis that is performed at each node of the decision tree is given below. As with all the trees presented in this report, this tree is used to iterate over a particular type of element in the UML diagrams. The classifiers in the system are made available through the `classifier` parameter.

- *Search Large Hierarchy:* Searches for a hierarchy of classes that is larger then the defined limit. Disregards classifiers that are not the root of a hierarchy (i.e. those that have super classes or implement interfaces). This is a very simple test designed to quickly eliminate smaller hierarchies, where using the *Decorator* pattern would be considered superfluous.
    - *in hierarchySize:Integer* – size of hierarchy over which should be considered.
    - *in classifier:Classifier* – iterate over all the classifiers in the system.
    - *in trivialGeneralizations:Set<Classifier>* - the set of classifiers that would be considered to be a trivial abstraction.
    - *out rootClassifier:Classifier* - the root of the hierarchy being investigated.

- *Similar Interfaces:* Checks that the interfaces within the hierarchy do not vary widely.
    - *in interfaceRatio:Double* – the ratio of inherited public methods versus the newly defined public methods that is considered to be a close compliance to the original interface.
    - *in rootClassifier:Classifier* - the root of the hierarchy being investigated.

- *Scan Hierarch for Independent Functionality:* Check to see if the hierarchy is completely balanced. This is a good indication that the functionality added at each level is independent as all possible combinations are being used. If this test fails we cannot be certain of the functional independence, thus we have to fall back on the user input.
    - *in rootClassifier:Classifier* - the root of the hierarchy being investigated.

- *Independent Functionality:* Query the user to determine if the functionality added through inheritance in the hierarchy can be considered independent.

- *Decorator Pattern:* Suggest that the *Decorator* pattern may be applicable to this structure.

For a more detailed formal description of the analysis using OCL is given in Appendix E.

6.4.3 Scalability

This algorithm will scale linearly $O(N)$, with respect to the number of classes in the system. The algorithm analyzes large hierarchies in the system, however these hierarchies must first be determined by the analyzing the descendants of each class. The algorithm also scales linearly with respect to the large hierarchies found in the system;

however this is more significant as many can lead to user interactions which are a performance concern.

6.4.4 Discussion

Use of this approach on real world systems will help determine the proper variable values for the hierarchy size (`hierarchySize`) and ratio of newly created public methods versus those inherited (`interfaceRatio`). The value of these variables is of significant importance because they play a key role in optimizing the accuracy and the completeness for identification of this candidate structure. They will be continually refined as more systems are explored.

7 VISITOR/ADAPTER PATTERNS

7.1 Visitor Pattern

7.1.1 Description

The *Visitor* pattern allows the designer to abstract a set of heterogeneous objects. It is an elegant manner to perform a common task on a set of differentiating objects without altering the classes themselves. The structure of this pattern is given in Figure 51.

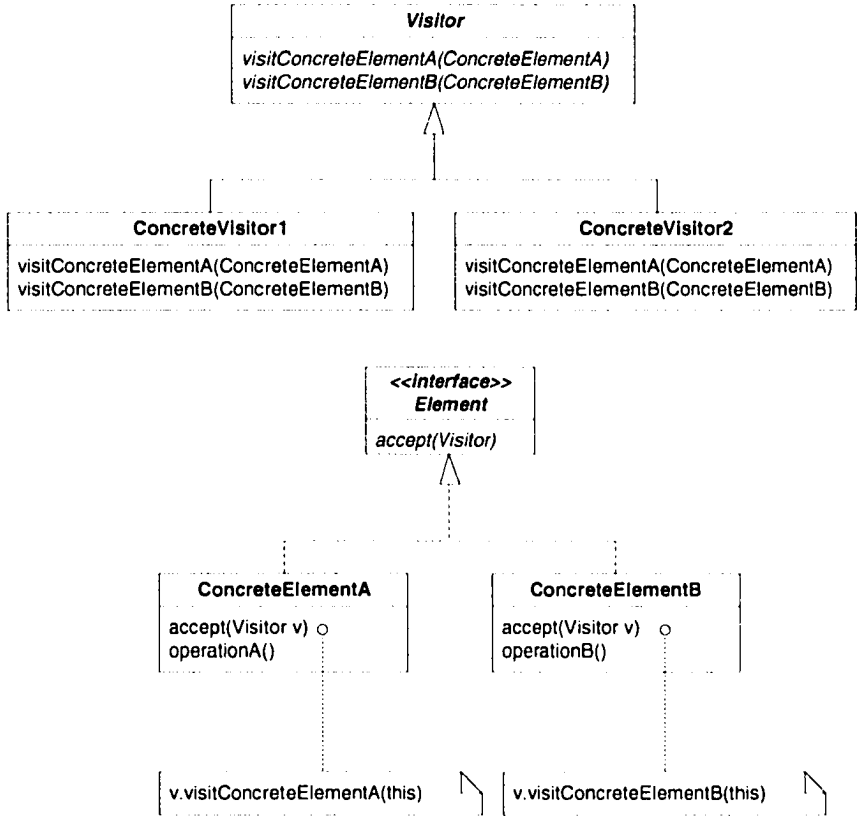

Figure 51 - *Visitor* Pattern Structure [26]

7.1.2 Applicability

There are several opportunities where this pattern can be applied.

- When dealing with a set of objects with different interfaces and the actions to be performed depend on the concrete type of the object.

- There are several distinct actions or analysis to be performed on a set of differing objects. The application of the *Visitor* pattern avoids polluting the classes with these operations and helps with maintenance as the code for each operation is in a single class.

7.1.3 Discussion of Candidate Structures

A method of dealing with a variety of heterogeneous objects is to test for the object's type (class/interface) and perform actions based on the type of object. This is done in Java by making use of the `instanceof` operator, which tests if an object can be typecast to the provided type. A developer may make use of a common if-then-else conditional structure and the `instanceof` in the conditions for each of the interaction operands. Such a structure is illustrated using UML in Figure 52.

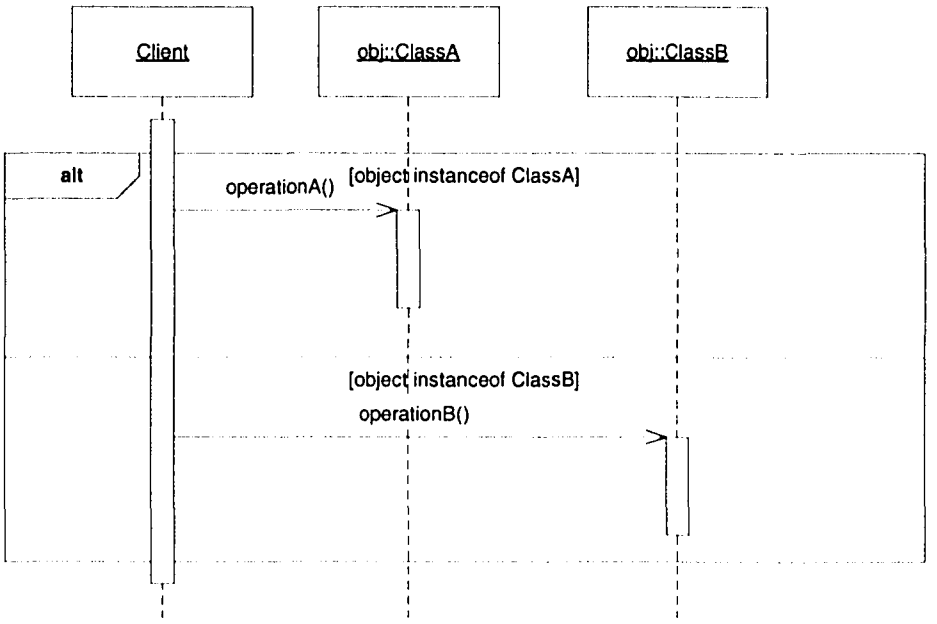

Figure 52 - Candidate Structure for *Visitor* Pattern

This structure is difficult to maintain as more objects are added to the set or if the operations to be performed within each interaction operands change frequently. As well this set of objects may be used in other areas of the software system to perform other operations requiring similar difficult to maintain conditional structures. Creating a *Visitor* interface to replace these structures allows for easier maintenance/addition of operations executing on this set of objects.

The limitation to the *Visitor* pattern is when the set of objects changes significantly and/or frequently. This will result in changes to the `Visitor` interface and if there are many concrete visitors hanging off this interface this may be a costly operation. As an example assume the *Visitor* pattern has been applied to a set of objects as illustrated in Figure 53. With this particular implementation a simple change to the set of objects would result in alterations to the `Visitor` class and all its subclasses. For a more detailed example please refer to the ATM Case Study in section 10.3.2.

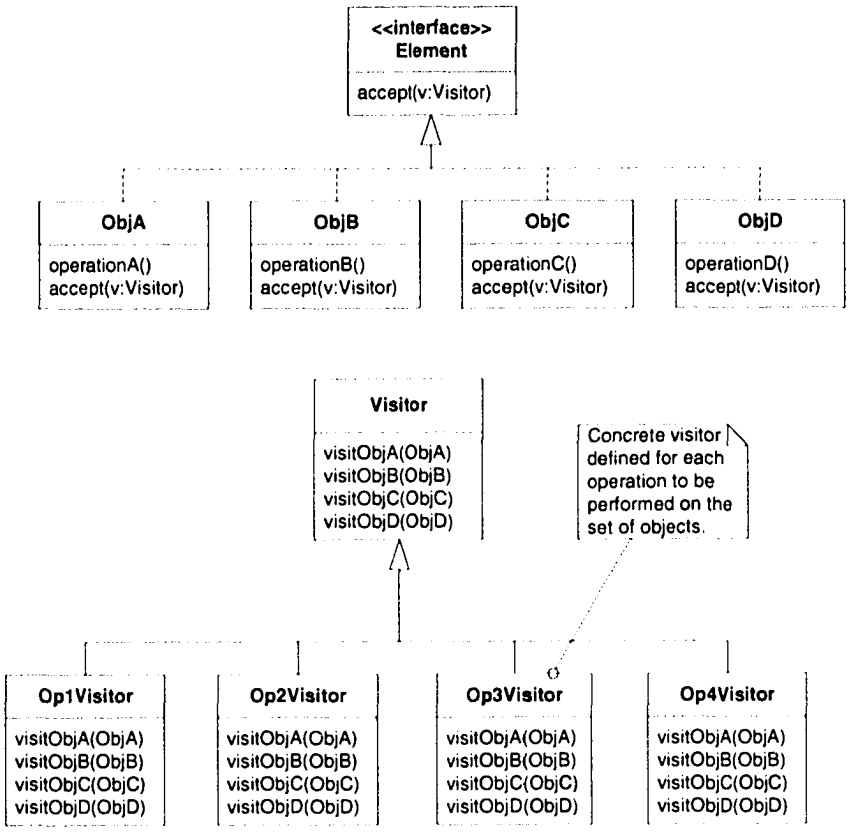

Figure 53 - *Visitor* Implementation

If it is a case where the set of objects frequently changes and there are many different operations being performed, it may be best to integrate the operations into the objects themselves as illustrated in Figure 54. However if the operations change in this structure it will result in alterations to four classes. This approach also suffers in that the operation code is spread through the four different objects; this may result in unnecessary code duplication.

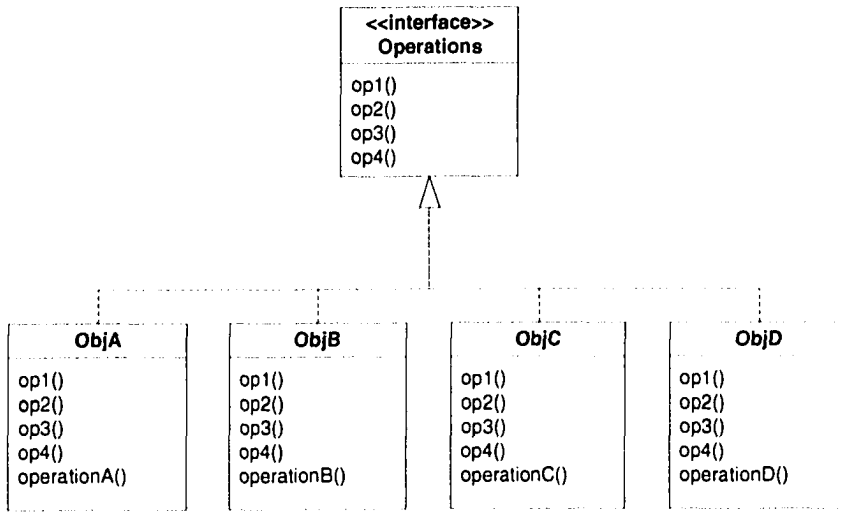

Figure 54 - Integrated Operations Implementation

For this same reason it is always best to define separate `Visitor` interfaces for each set of objects. If the designer attempts to create a single `Visitor` interface to encompass several different sets of objects, then a change to any of the sets may result in a change to the `Visitor` interface and the implementations of that interface.

7.2 Adapter Pattern

7.2.1 Description

The *Adapter* pattern is simply used to transform a given interface to one that is more suitable for the developer's use in the application. The *Adapter* acts as a translator between classes. There are two different versions of the *Adapter* pattern: the class adapter and the object adapter. Both versions are given in Figure 55 and Figure 56.

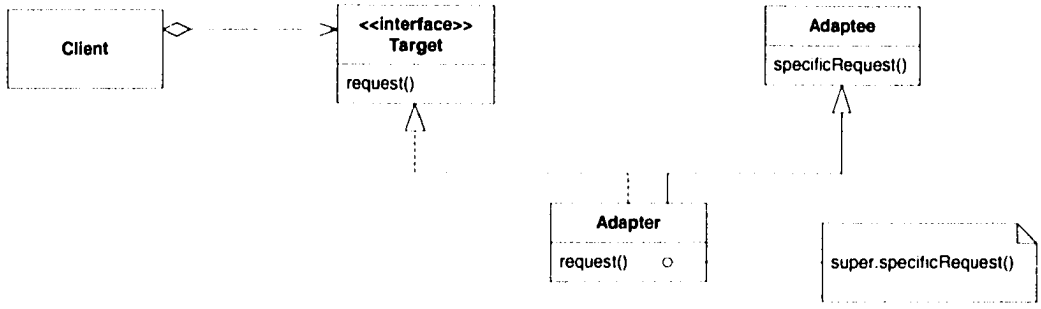

Figure 55 - Class *Adapter* Pattern Structure[26]

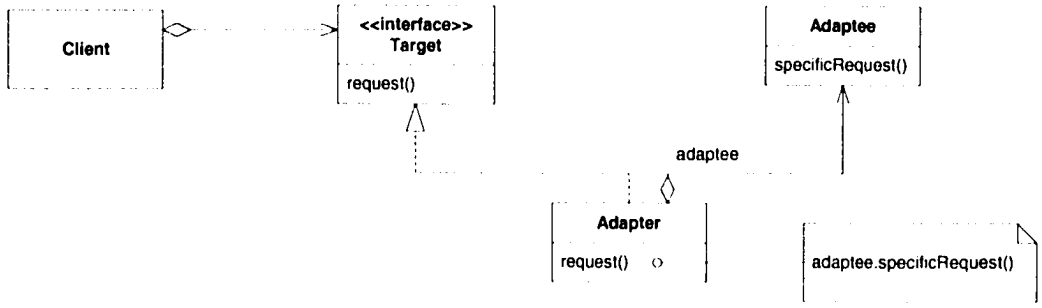

Figure 56 - Object *Adapter* Pattern Structure[26]

The choice of which to use is based on the circumstance and the developer's preference.

7.2.2 Applicability

The applicability of this pattern always involves mismatched interfaces between the client and the server classes [26].

- The use of an existing class where the interface does not match that desired.
- The creation of a class to be reused by many different clients with different interface needs.
- The object adapter can be used to adapt a root class in a hierarchy as to be able to use any of the subclasses with the desired interface.

7.2.3 Discussion of Candidate Structures

The candidate structure for this pattern closely resembles the candidate structure for the *Visitor* pattern given in section 7.1.3. The candidate structure is again the alternative combined fragment structure with the guard conditions on the interaction operands containing a test for the object's type; this structure is illustrated in Figure 52. The only difference between the two structures is the purpose of testing the object's type. In the *Visitor* pattern's candidate structure the object's type is tested as to perform a specific operation based on the object's type. The operation is different for the various types or the if-then-else structure would not be required. In the case of the *Adapter* pattern's candidate structure the alternative combined fragment structure is used to test the object's type as to determine its interface. In this case all the various possible types (classes/interfaces) provide similar functionality; the conditional test is to invoke this functionality on the diverse interfaces. A good example would be a peer to peer messaging application that uses a variety of protocol (MSN, ICQ, email, etc.) and corresponding libraries to send messages. There are various networks and protocols that allow user to communicate to one another, assuming that there are libraries to make use of each protocol. This example is illustrated in Figure 57. The client may test the type of the object as to determine how to invoke on that object's interface.

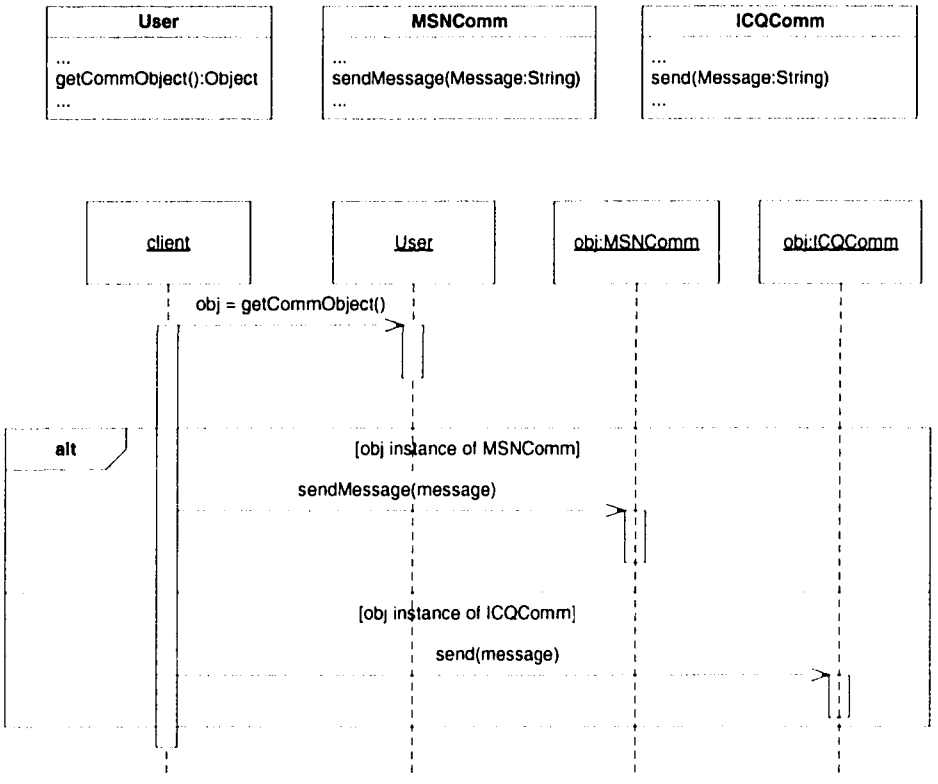

Figure 57 - *Adapter* Pattern Candidate Structure

This particular design can be simplified by using a common interface and the *Adapter* pattern. The static structure for this alteration is given in Figure 58.

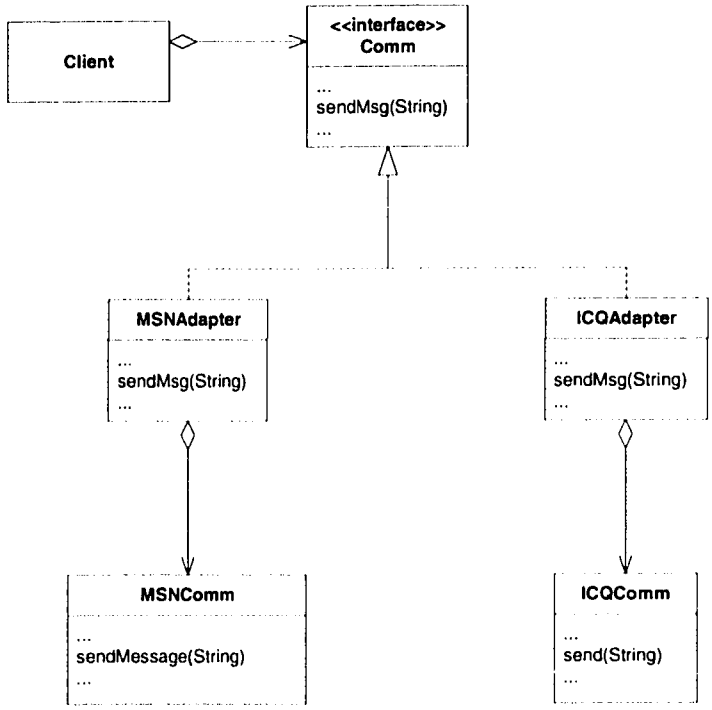

Figure 58 - *Adapter* Pattern Applied

Unfortunately the difference between whether the type checking is performed to determine the interface or to determine the proper operation to be performed cannot be easily determined using automatic analysis. The only option would be analyzing the OCL constraints of the methods invoked on the objects in the interaction operands to determine if the functionality is similar. This is beyond the scope of this research project; therefore we must rely on the user to provide this information.

7.3 Identification of Candidate Structures for Visitor/Adapter Patterns

7.3.1 Meta-Model

In order to identify the candidate structures for these patterns the meta-model presented in Figure 36 must be expanded to model the guard condition on some dynamic elements. These enhancements are illustrated in Figure 59, removing the unneeded information from the meta-model.

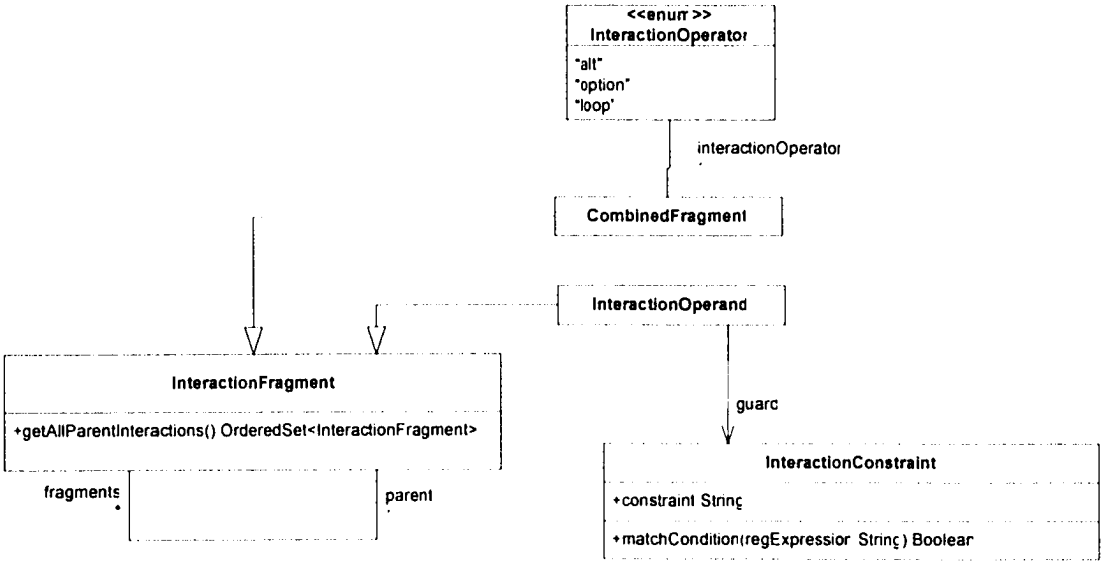

Figure 59 - Meta-Model for *Visitor* and *Adapter* Patterns

A new element is added to the meta-model, `InteractionConstraint`, to model the logical constraint that guards the execution of the interaction operand. The `constraint` attribute of the `InteractionConstraint` is the guard condition which must be satisfied for the execution of this set of interactions. The `matchCondition` method check if the `condition` string matches the regular expression provided. Note that the format for regular expressions used in this report is the Java format presented in [48].

7.3.2 The Visitor and Adapter Decision Tree

The decision tree for the candidate structures for these design patterns is given in Figure 60.

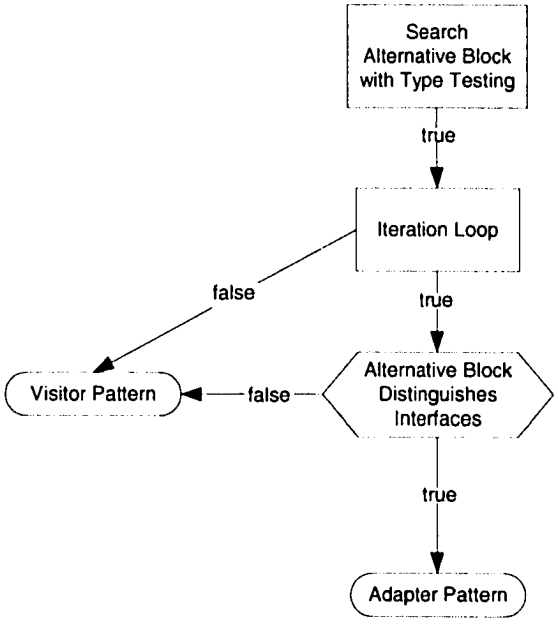

Figure 60 - *Visitor* and *Adapter* Pattern Decision Tree

The description of the analysis performed at each node is given below.

- *Search Alternative combined fragment with Type Testing:* Search for alternative combined fragments in the system where the conditions for the alternatives contain some form of test for an object's classifier type. This check is performed by examining the condition for the presence of the `instanceof` expression (note that this would be a Java dependent implementation).
    - *in combFrag:CombinedFragment* - the alternative combined fragment being investigated.

- *Iteration Loop:* Check to determine if the alternative combined fragment is found within a loop structure. If the alternative combined fragment is found within a loop structure the condition is analyzed to determine if the loop is likely used as to iterate over a set of objects. This test will perform checks for key words within the condition of the loop, such as: 'for each', 'size', 'hasNext', etc.
    - *in combFrag:CombinedFragment* - the alternative combined fragment being investigated.
    - *in keywords:Set<String>*- the set of keywords that can be used to test the condition for iteration.

- *Alternative combined fragment Distinguishes Interfaces:* Query to the user to determine if the alternative combined fragment is mainly used to determine the proper method signature to invoke on the object.

- *Adapter Pattern:* Suggestion to explore the use of the *Adapter* pattern to replace the alternative combined fragment.

- *Visitor Pattern:* Suggestion to explore the use of the *Visitor* pattern to replace the alternative combined fragment.

7.3.3 Scalability

This algorithm scales linearly, $O(N)$, with respect to the number of combined fragments found in the UML model. More significant is that this algorithm requires user input for combined fragment that pass the first two conditions. The user interaction is expensive, and should be avoided. Testing will be required to determine the commonality of alternative combined fragments which test as object's type in the guard conditions; if these are found to be common there will have to be further analysis added to this decision tree to avoid the user interaction.

7.3.4 Discussion

The decision tree generated for these candidate structures is fairly simplistic and relies heavily on the user's input. The use of the classifier type checking is a clear sign that a design pattern can be used to provide a more elegant solution; however as mentioned before it is difficult to determine why the type check is used initially. This information is important in determining the proper design pattern to be applied. The check for the iterator loop structure is a good indication that the type check is performed as part of an iteration over a set of heterogeneous objects. The check for the iterator loop may however be difficult as there are many different expressions that may be used in the condition. Also the choice to only search the direct parent for the loop structure may be limiting; however at this time it seems that this would cover the majority of the cases. More research during the case study phase of the research may prove to the contrary.

8 STATE PATTERN

8.1 Description

The state of any object is depicted by the values of its attributes. For some objects the current state of the object dictates its behaviour, these objects are referred to as stateful objects. Some classes are heavily state dependent with many different states and state dependent behaviour that can vary widely. In this case the *State* pattern provides an efficient methodology for creating such a class. The structure of the *State* pattern is illustrated in Figure 61.

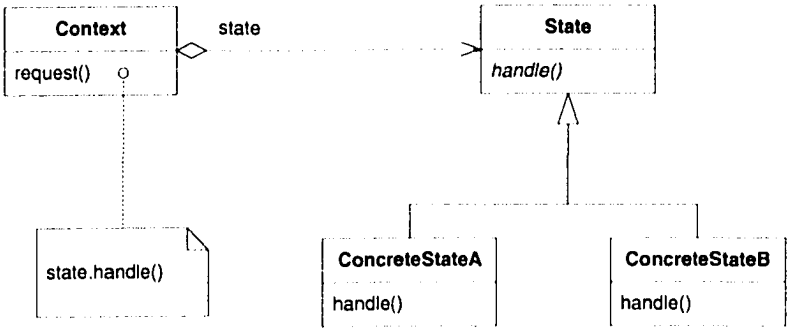

Figure 61 - *State* Pattern Structure [26]

The UML standard uses Statecharts to model an object's states and behaviour at each state. The *State* pattern shown here does have several limitations when used to implement a UML Statechart [7].

- Difficult to implement actions on the transitions between states.
- No means for implementing the entry/exit actions associated with the states.
- Activities to be performed in the state; there may be some code duplication if similar activities are performed in different states.

For these reasons the *State* pattern needed to be expanded. The *Extended State* [7] pattern eliminates some of the deficiencies of the *State* pattern, but at the cost of a more complicated pattern structure as illustrated in Figure 62

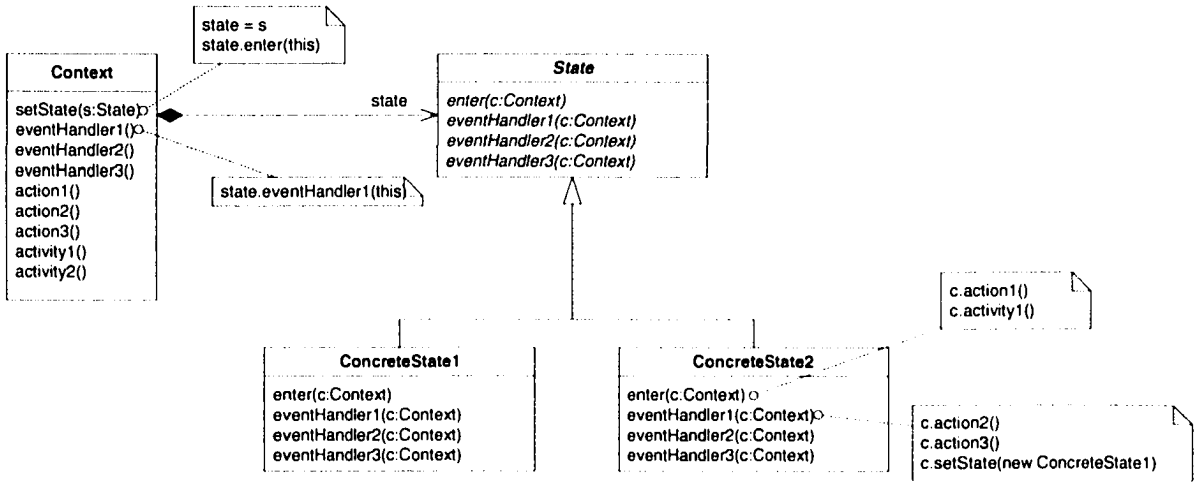

Figure 62 - Extended *State* Pattern Structure [7]

The choice of using the *Extended State* pattern in place of the *State* pattern is based on whether the need for the modeling entry/exit events, or whether states have similar actions to be performed. If this is not the case, then the simplicity and ease of implementation make the *State* pattern a more logical choice.

8.2 Applicability

The state pattern is mainly used when an object's run-time behaviour depends on its state.

8.3 Discussion of Candidate Structures

The candidate structure of this design pattern explores the use of an object's state for run-time decision making. Therefore the candidate structure is composed of alternative combined fragments that use attributes of the class in the conditional predicates or guard conditions. There are conditions for the logical expressions and the attributes that need to be addressed: the attributes should be primitive, the attribute should be used in each

predicate found in the conditional structure, and there should be more then one instance where this attribute is used to base decisions for the behaviour of the class.

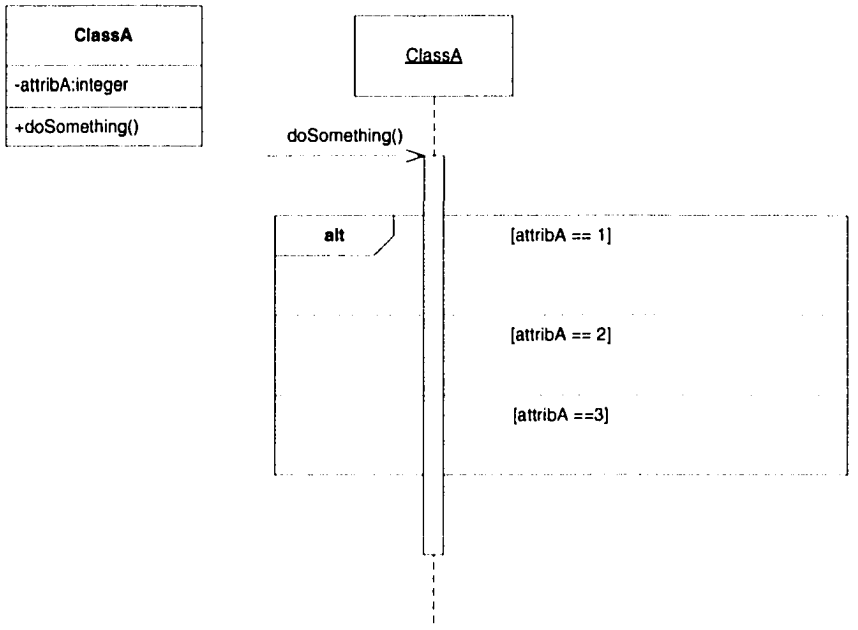

Figure 63 - *State* Pattern Candidate Structure

To ascertain if an attribute is used in the guard conditions simple pattern matching is performed based on the class' attributes names. A more elaborate scheme could be devised; however this would require more computation during the search.

8.4 Identification of Candidate Structures for the State Pattern

8.4.1 Meta-Model

The meta-model containing the information required to accurately identify the candidate structures for this pattern is given in Figure 64. The structures are similar to those found in previous meta-models with the exception of the new association between `CombinedFragment` and `Object`. This association designates the object whose behaviour is controlled by the `CombinedFragment`.

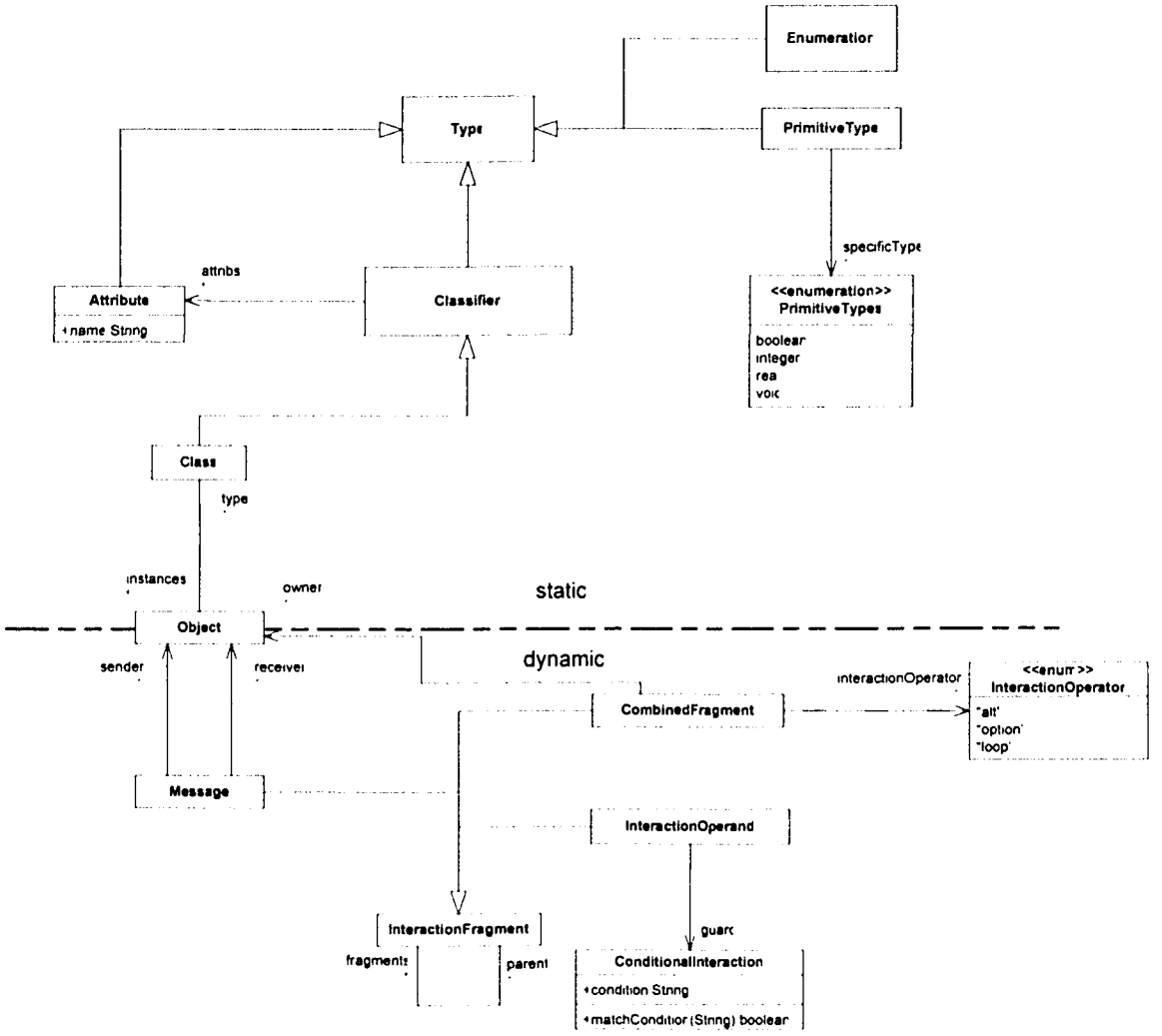

Figure 64 - Meta-Model for *State* Pattern

8.4.2 State Pattern Decision Tree

The decision tree developed to identify the candidate structures for the *State* pattern is given in Figure 65.

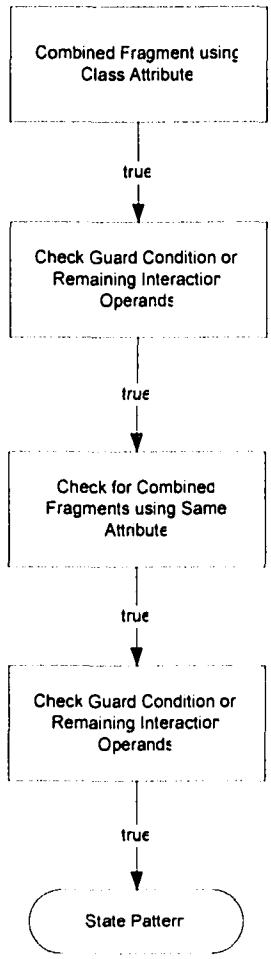

Figure 65 - Decision Tree for *State* Pattern

The formal logical description of the analysis performed at each node is given in Appendix F. The informal description of the analysis performed at each node is given below.

- *Combined Fragment using Class Attribute:* Locate alternative combined fragments where the first interaction operand makes use of a primitive attribute in its guard condition (interaction constraint).
    - *in combFrag:CombinedFragment* - the alternative combined fragment being investigated.

- *out cf2bChecked:CombinedFragment* – an alternative combined fragment to be checked as to determine if each of the interaction operands use the same attribute in their guard condition.
- *out attribute:Set* – the attribute used in the first guard condition.

- *Check Guard Condition on Remaining Interaction Operands:* Check performed to determine if the primitive attribute is used in each of the guard conditions for the interaction operands composing this alternative combined fragment. This check is used twice in the analysis for the candidate structures.
  - *in cf2bChecked:CombinedFragment* – an alternative combined fragment to be checked as to determine if each of the interaction operands use the same attribute in their guard condition.
- *Check for Alternative Combined Fragments using Same Attribute:* Check performed to determine if the same primitive attribute is used in other alternative combined fragments to determine the run-time behaviour of the class.
  - *out cf2bChecked:CombinedFragment* – an alternative combined fragment to be checked as to determine if each of the interaction operands use the same attribute in their guard condition.

8.4.3 Scalability

The worst case scenario for this algorithm is $O(N^2)$ with respect to the combined fragments found in the system. Similar to the decision tree for the *Abstract Factory* and *Factory Method*, this performance can be improved by filtering the combined fragments that make use of attributes in logical expression for each of the interaction operands. This would reduce the number of combined fragments to cross compare. Further scalability testing would have to done to determine if these steps will be required to improve performance.

8.4.4 Discussion

The approach developed to locate this candidate structure assumes that the state of the object is determined by a single primitive or enumeration type attribute. There may be cases where the state is governed by several attributes, each playing the role of state variable depending on the circumstances. This situation is somewhat dealt with by creating a set of attributes used in the guard conditions in the first step ('Combined Fragment using Class Attribute') of the decision tree. The generated set is used to locate other alternative combined fragments which make use of at least one of the attributes found in the set.

9 DESIGN

This section outlines the design of the Design Pattern Analysis tool (DPATool) built as a proof of concept of the approach discussed in the previous sections. The DPATool consists of three subsystems as illustrated in Figure 66: the `DPA Eclipse PlugIn`, the `DPA Processing Engine`, and the `DPA Model`. DPATool is built as a plugin to the Eclipse platform and interacts with two other Eclipse projects, namely `UML2` and `EMF`.

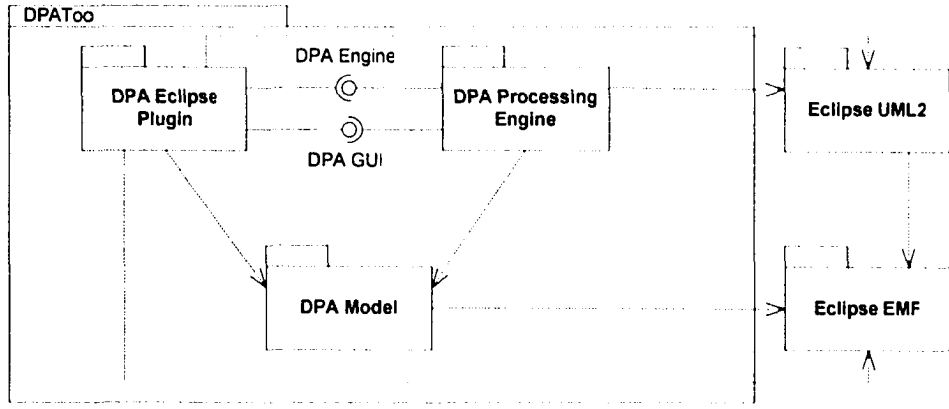

Figure 66 - DPA Architecture Overview

The `DPA Processing Engine` is the main subsystem of the tool, responsible for the analysis of the UML models. The UML models to be analyzed are instantiated using the the Eclipse UML2 project [21]. The Eclipse UML2 project is an open-source implementation of the UML 2.0 metamodel [41] and has been integrated into some industrial case tools such as IBM's Rational Software Architect (RSA) [44] and Omondo's EclipseUML Studio [42]. The UML2 project is based on the Eclipse Modeling Framework (EMF) [13], which will be further discussed in section 9.1.1.

The `DPA Model` subsystem is used to model the decision trees discussed in previous sections. The models are then used by the `DPA Processing Engine` to guide its analysis of the UML2 model.

The `DPA Eclipse Plugin` or the platform subsystem is the client invoking the `DPA Processing Engine`. The `DPA Processing Engine` is designed as a library component that can be integrated into a wide variety of case tools, such as those mentioned previously. For the purpose of this project the DPATool has been integrated into the Eclipse Platform as a plugin. The fact that the Eclipse Platform is the foundation for both the RSA and the EclipseUML Studio motivated the choice to implement the DPATool prototype as an Eclipse plugin.

The following sections detail further the design of the three subsystems that comprise the DPATool. We start with a short description of the Eclipse platform and its related projects that we use as this drove some of our design decisions.

9.1 The Eclipse Platform

Eclipse [14] is an open source project geared to providing a robust, full-featured, industry platform for the development of highly integrated tools and client applications. The goal of the Eclipse project is to eventually provide an industry standard platform.

The Eclipse project contains within itself many subprojects or categories: The Eclipse Project, The Eclipse Tools Project [17], The Eclipse Technology Project [15], The Eclipse Web Tools Platform Project [18], The Eclipse Test and Performance Tools Platform Project [16], Business Intelligence and Reporting Tools Project [12]. These subprojects are further divided into smaller projects. The two categories of interest for this thesis are the Eclipse Project and the Eclipse Tools Project.

The Eclipse Project contains the main components of the Eclipse platform: the platform framework, Java Development Tools [19] and the Plug-in Development Environment [20]. These tools were used to build the initial prototype of the DPATool and are used in the execution of the tool.

The Eclipse Tools Project contains several smaller projects not essential to the Eclipse Platform, but can be seen as value added feature. Two of these projects were critical in

the implementation of the DPATool, namely the Eclipse Modeling Framework [13] (Section 9.1.1), and the UML2 Project [21].

The Eclipse Platform provides the foundation from which many commercial tools have been developed (i.e. IBM Rational Software Architect [44], Omondo's Studio [42]). The DPATool makes use of the OCL engine (Section 9.1.2) which is a component of the Rational Software Architect from IBM.

9.1.1 Eclipse Modeling Framework (EMF)

The Eclipse Modeling Framework (EMF) [13] is a modeling and code generation engine for building applications based on a structured data model. It allows developers to define a data model, under the form of a UML class diagram (the class diagram does not describe any operation though as this is only a data model). Such a data model can be for instance specified with a case tool such as Rational Rose [27](which is the case of the DPA model described in following sections). The data model is used by the EMF facility to create a set of Java classes for the model, a set of adapter and visitor classes to enable viewing and visiting any instance of the data model, and a set of classes that can be used as a basic editor for the data model. Additionally, the Java classes generated by EMF provide the following functionalities [27]:

- Built-in persistence support for the model, including XMI serialization. Note that the XML schema for a given model is generated automatically.
- Model change notification feature: The developer can create classes that would be informed (event listeners) when instances of data model classes are modified (e.g., by an editor).
- Reflection API, more powerful than the standard Java reflection API, to allow EMF objects to be manipulated generically.
- Interoperability with other EMF-based tools and applications.

- Extensive library to aid in the development of applications which visualize, create and modify the model (editor).

- A generic editor for the model can be created.

It is important to note that the data model is built without any programming language in mind for its implementation, although the end result is a set of Java classes: It is simply a data model. In particular, the data model can make use of multiple inheritance, although multiple inheritance is not supported in Java. The EMF facility is able to account for that.

9.1.2 The OCL Engine

Along with EMF, IBM has defined an OCL Engine to gather information on an instance of an EMF model under the form of OCL queries.

9.1.2.1 General introduction to the OCL engine

The OCL engine accepts an OCL statement (i.e. String object) to be evaluated against an EMF object. The OCL engine makes use of an ANTLR generated parser to parse the OCL expression and create the AST tree. The *Visitor* pattern is then used to visit each node of the AST generated as to validate the tree. The AST tree is also used to guide the evaluation of the OCL expression, by visiting each node of the tree and performing the require operations on the UML model.

The EMF framework provides strong reflection abilities into the models it generates. This reflection capability is used by the OCL Engine to evaluate the OCL expression against an object. The reflection capabilities of EMF allow for associations to be traversed and methods to be invoked on the object in question. The types and methods defined in the OCL specification (OclAny, Set, Sequence, etc) are represented by adapters in the OCL engine.

9.1.2.2 Improvements

The use of OCL expressions allow the decision tree developer to quickly define new analysis to be performed on the UML2 model; however OCL has its limitations. As an example, the OCL specifications lack the ability to perform recursive functionality. This limits some of the analysis that can be performed on an arbitrary UML2 model. To deal with the limitations of OCL, two different solutions have been implemented. Firstly, the OCL Engine has been enhanced to allow for the introduction of new methods in the model being analyzed.

The model extension feature was developed as to reduce the complexity of the decision trees developed. During the development of the decision trees before the introduction of this feature, there was a large number of Java nodes needed in the decision tree. A cleaner solution of providing a means to extend the EMF (in this case the UML2 model) model in the OCL Engine was proposed. This would allow for the flexibility of the Java language, with the ease and comprehensiveness of OCL expressions.

To implement this feature a new interface was introduced, `ModelExtensions`. The OCL Engine subsystem accepts objects which conform to this interface. When an OCL expression refers to a method that is not found on the EMF model, the OCL Engine then makes use of the Java Reflection API [47] to search the provided `ModelExtension` objects for the required method. To make the search possible a special signature must be used for the methods. The required signature is formed from the name of the class and the name of the method being invoked. As an example, an operation is to be added to `InteractionOperand` to check the guard condition against a regular expression. The operation name is to be `matchGuardCondition`, and the operation must accept a string containing the regular expression to be used. The method on the `ModelExtension` object is given in Figure 67.

```
public boolean org_eclipse_uml2_InteractionOperand_matchGuardCondition
      (InteractionOperand io, String regExp){
```

```
    Expression exp =
      ((Expression)io.getGuard().getOwnedElements().get(0));
    String body = exp.getBody();
    return body.matches(regExp);
  }
```

Figure 67 – Example Method from ModelExtension

Note that the first argument of the method is the object on which the method is to be invoked. This method can now be used in an OCL expression such as the example given in Figure 68.

```
self.operand->forAll(io:InteractionOperand|
(io.matchGuardCondition('.*[a-zA-Z_0-9]+  instanceof  [a-zA-Z_0-9]+.*'))
or (io.matchGuardCondition('.*else.*')))
```

Figure 68 – OCL Statement Using Model Extensions

In the prototype version of the DPATool the `ModelExtension` classes were implemented manually; however their format lends well to the development of more sophisticated and user friendly tools.

The other improvement to the OCL engine was the ability to use more than one EMF (the context) object in the OCL expression. This was necessary to perform comparisons between the EMF object in the UML model. As an example assume that some decision tree has identified a class (say `classA`). It is then necessary to determine if `classA` is the super class of `classB` (also identified earlier in the model). With the improvement to the OCL engine this is possible by making use of parameters to the OCL expression (discussed in futher detail in section 9.2.3). The resulting OCL expression is simply, `'self.super = classA'` using `classB` as the context of the OCL expression.

9.2 The `DPA Model`

We describe the `DPA Model`, which can be referred to as the decision tree metamodel, in four steps. We first present the general structure of the decision tree (i.e., its nodes) as a set of related classes (Section 9.2.1). We then describe how queries on the model are

represented in the tree metamodel (Section 9.2.2). Section 9.2.3 shows how data can be exchanged between nodes, for instance a node producing information that is used as an input by another node. Section 9.2.4 describes how information on design patterns is represented in the metamodel, and Section 9.2.5 shows an example of instantiation of the decision tree metamodel for an example decision tree. Three different UML models are referred to in this section: the UML 2 standard metamodel [41], referred to as the *UML2 metamodel*; the decision tree metamodel, referred to as the *metamodel*; and the UML 2 model of the system to be analyzed, simply referred to as the *model*. Note that the metamodel is introduced in a step wise manner, each time focusing on one specific aspect. The complete metamodel can be reconciled from Figure 69, Figure 70, Figure 72, and Figure 74. Last, we show how a decision tree can be built (i.e., the metamodel instantiated) using the decision tree editor generated from the model by the EMF framework (section 9.1.1).

9.2.1 General structure

The decision tree metamodel consists of the `DPADecisionTree` class which provides the identity of the decision tree (attributes `name` and `description`), as shown in Figure 69. A decision tree is composed of a set of nodes (abstract class `Node`), one of which is the root of the tree. The `DesignPattern` class represents the design patterns that are applicable to the candidate structures that can be identified by this decision tree.

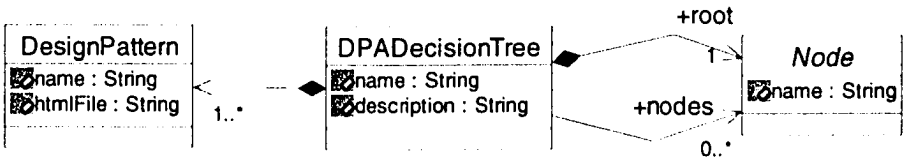

Figure 69 - DPADecisionTree and Related Classes

As illustrated in previous sections, decision trees show different kinds of nodes. The corresponding classes in the decision tree metamodel, child classes of `Node`, can be found in Figure 70. As can be seen from Figure 70, the subclasses of `Node` have been categorized by the number of paths leading from that node. The `UnaryNavigationNode`

represents nodes which have a single path to follow, while the `BinaryNavigationNode` has two paths (a true and a false path). Another kind of node is the `OrConditionNode` which has several paths to follow and represents the 'or' condition found in the decision trees. Last, the `DesignPatternNode` is the leaf of the decision tree, and represents a suggestion to the user for the design pattern to be applied to the UML elements identified by the decision tree analysis . (Association between `DesignPatternNode` and `Node`, with rolename `resetNode`, is not intended to describe leading paths in the decision process, as for associations between the other subclasses of `Node` and `Node`. Association with rolename `resetNode` is further described in Section 9.3.) The other classes are described in the following section.

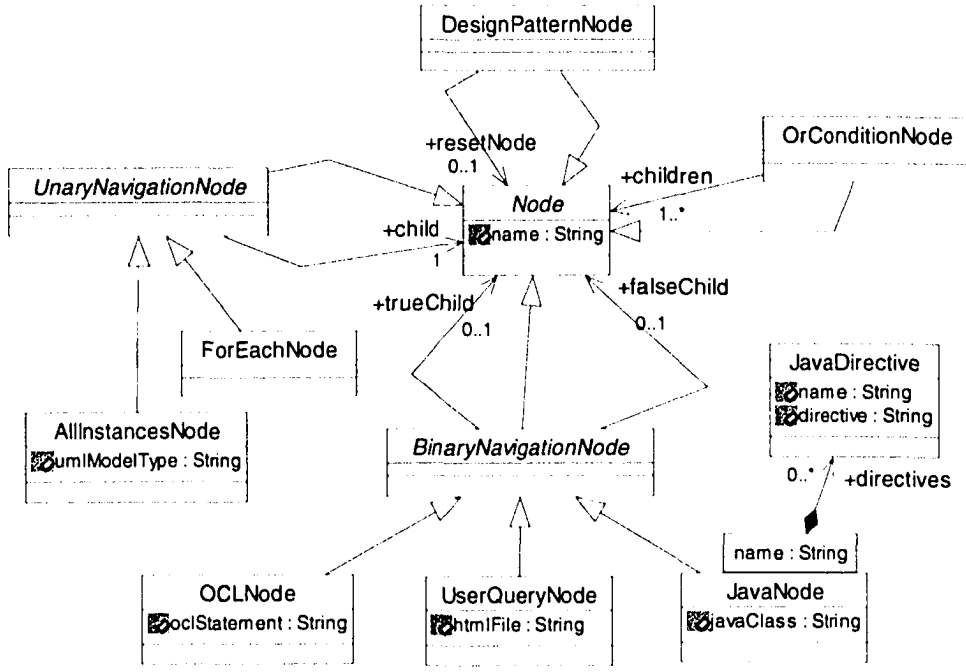

Figure 70 - Node Navigation Model

9.2.2 Handling model queries

During the decision process, decisions are taken according to structures that can be found in the model under investigation: e.g., interaction operands showing messages sent to objects in an inheritance hierarchy (section 4). As described previously, these structures can be specified using OCL expressions, and these OCL expressions can be used to query

the model, instance of the UML2 metamodel. Fortunately, there exists an OCL engine to perform OCL queries on an EMF based metamodel (section 9.1). However, this engine does not support OCL keyword `allInstances` [40], and has other limitations with respect to our work (section 9.1). Last, some issues in the decision process cannot be resolved by an automated tool and the user input is sometimes required. Addressing these issues is the purpose of classes `AllInstancesNode`, `ForEachNode`, `OCLNode`, `JavaNode`, `JavaDirective`, and `UserQueryNode`.

Most decision trees developed using the `DPA Model` start with the identification of all the occurrences of a given structures that possess certain properties. In other words they begin with the identification of all model elements of a given type followed by a set of constraints to determine if this element meets our given structure. In OCL terms, they start with an `allInstances`, which is represented as the `AllInstancesNode`. The `ForEachNode` is used to iterate over the elements in a list and process the items individually. The `OCLNode` allows the designer of the decision tree to formulate an OCL expression to be evaluated against a UML element from the model: such OCL expressions can be evaluated directly using the OCL engine. Then, in a typical decision tree, an instance of `AllInstancesNode` is followed by (association between `UnaryNavigationNode` and `Node` with rolename `child`) an instance of `ForEachNode`, which is itself followed by an instance of `OCLNode` (see an example instantiation in Section 9.2.5). As described in Section 9.3, the `ForEachNode` followed by the subsequent processing is then in charge of verifying whether the model elements identified by the `AllInstancesNode` possess the required characteristics to identify a potential pattern usage.

When necessary, that is when OCL does not allow the designer to efficiently describe a query of the model, classes `JavaNode` can be used. A class containing the Java code to be executed (the query) using dynamic loading during the decision process to provide the greatest amount of flexibility. The class to be loaded is uniquely identified by its name, which is the attribute of `JavaNode`. As further described in Section 9.3, the dynamically

loaded class is expected to implement a specific interface, `JavaNodeImpl`. `JavaNode` can be considered a placeholder for Java classes that the designer implements to perform queries on the model. The `JavaDirective` represents arguments potentially used by the Java class to change the operation of the query. They allow the Java classes to be implemented more generally and used in a variety of decision trees. The Java language and the use of the UML2 metamodel API provide the most flexible means of automated analysis. This flexibility comes at the cost of ease of development and comprehensibility that the OCL language offers.

As an example, OCL has no means of performing regular expression matching on a string variable; therefore a solution would be to develop a `JavaNode` to perform this analysis. The regular expression is to be checked against the guard conditions of a `CombinedFragment`, this combined fragment is referenced by the npl object (NodeParameter is described in section 9.2.3). By using the `JavaDirective` class, a generic `JavaNode` can be developed to test the guard conditions of `CombinedFragments`, as shown in Figure 71. This `JavaNode` can now be reused in a variety of decision trees where the regular expression will be different as illustrated in Figure 71.

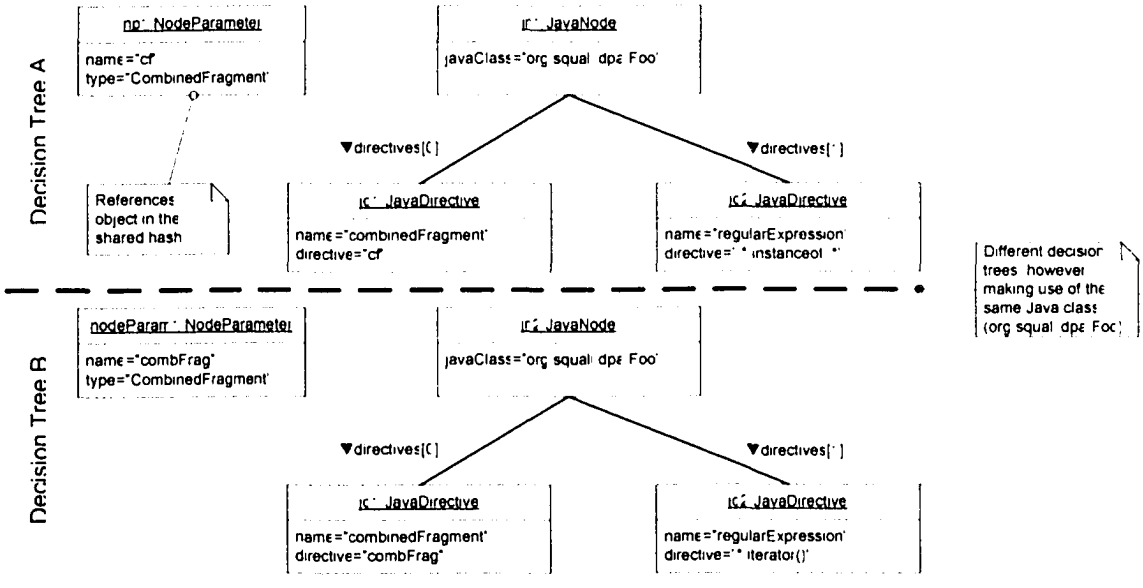

Figure 71 - JavaNode and JavaDirective Intanstiation

Last, when neither an OCL expression nor a piece of Java code can provide an answer to a query (i.e., the user input is required) the `UserQueryNode` class can be used in a decision tree. It allows the decision tree developer to query the user, by means of an HTML document, for information that cannot be gathered automatically. The result from the user (true or false) is then used to guide the navigation to the next node.

9.2.3 Exchanging data between decision tree nodes

There is a need to exchange information between nodes of the decision tree: i.e., the result of the analysis performed in a node can be an input to the following node(s) in the tree. (Note that when a node produces some data, the data are not necessarily used in the immediately following node, i.e., linked to the producer node with a child association in the metamodel, but can be a node further down in the tree.). A typical example concerns classes `AllInstancesNode` and `ForEachNode`: An instance of `AllInstancesNode` collecting a list of model elements (from the model under study) is followed by an instance of `ForEachNode` that iterates over the list to perform some query (defined in a `OCLNode` instance). Such data exchanged between decision tree nodes is represented in the metamodel by the `NodeParameter` class (Figure 72).

A `NodeParameter` instance is uniquely identified by its name (attribute `name`). It is a surrogate for an instance of one UML2 metamodel class (in which case attribute `type` is that class name) or a collection of UML2 metamodel class instances (in which case attribute `type` reads `Set(className)` or `Bag(className)`). (This facilitates type checking during the decision process.) These values for attributes `name` and `type` are provided by the designer of the tree, but do not match to any model element until an actual processing of a model occurs. During such decision tree processing (see Section 9.3), a mapping between the `NodeParameter` instances used in the decision tree and actual model elements is maintained in a hash table: keys are the names of the `NodeParameter` instances, and values are references to UML2 metamodel class instance(s). At decision tree processing time, a `NodeParameter` instance can be seen as a placeholder for (a set of) model element(s). This hash table plays the role of a global data

structure (global variable) and is accessible to all the nodes which need to exchange information, and the nodes get the exact information they need thanks to the keys: the nodes that need to exchange data know precisely which data in the hash table to exchange as data are named and typed.

Figure 72 also shows that not all nodes (child classes of `Node`) exchange data, and that nodes can produce data (class `ResultGenerator`), consume data, or both. An instance of `AllInstancesNode` collects (produces) all the instances of a given UML2 metamodel class (attribute `umlModelType`) found in the model under study. An instance of `ForEachNode` uses a collection of UML2 metamodel class instances (e.g., generated by an `AllInstancesNode` instance). A `JavaNode` instance can also use and produce data. Therefore a single `JavaNode` can very easily perform the functionality of several `OCLNode`. Although, as opposed to the other classes, the metamodel does not show explicitly (by means of an association) that `JavaNode` can use `NodeParameters` as inputs, this is the case: Written in Java, `JavaNode` can have access to all the `NodeParameter` instances, as well as the hash table holding them[7].

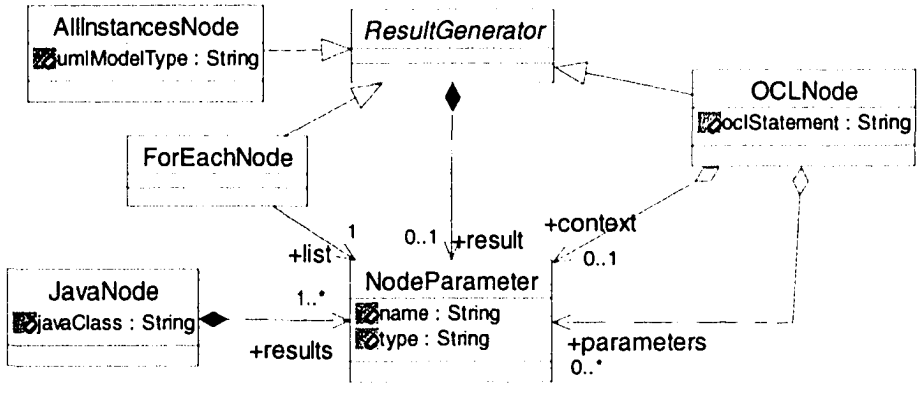

Figure 72 - Node Parameter Model

[7] One possible use of `JavaNode` is for debugging purpose. When designing a decision tree, the designer may want to add a `JavaNode` instance in charge of displaying the contents of the hash table, and thus observe the processing involved in the decision tree.

An `OCLNode` instance holds an OCL expression (attribute `oclStatement`) that is evaluated in a given context (association with `NodeParameter` with rolename `context`), which matches the OCL notion of context for OCL expressions. The context denotes the model element(s) on which the OCL expression should be evaluated. The `parameters` association objects specify additional model element(s) to be used as parameters to the OCL expression. The introduction of parameters into the OCL statements allows for greater flexibility in the decision tree, as multiple UML elements identified previously can be used within the OCL statements. As an example, assume that in a decision tree, it is required to determine whether a class (say `class2`) is a subclass of another class (say `class1`), but defines a new operation (say `op1`), and that the two classes and the operation have been identified by previous nodes in the decision tree (i.e., there are entries in the hash table for NodeParameter instances which names are `class1`, `class2` and `op1`). The ability to use parameters allows this problem to be modeled as shown in Figure 73.

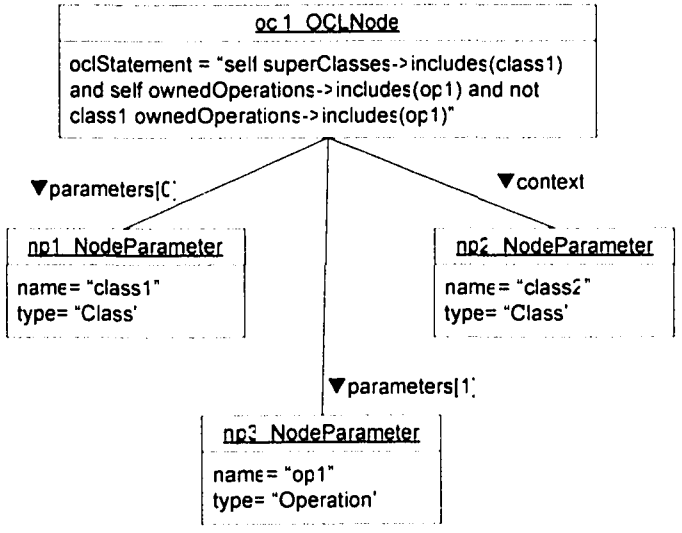

Figure 73 - Instantiating an OCLNode

Recall that an `OCLNode` is a `BinaryNavigationNode` (Figure 70). When the result of the OCL expression is a `boolean`, its value is used to determine which path to follow. When the result is an object or list, the state of the object or list (e.g., size of the list) is what determines the navigation. If an object is expected and the OCL Engine returns a `null`

result from the OCL expression or if the list returned is empty, the `falseChild` path is followed. The result returned from the OCL expression can also be made available to subsequent node by associating a `NodeParameter` object to the `result` association (Figure 72).

9.2.4 Describing found patterns

The leaf nodes of the decision trees are `DesignPatternNode` instances. A `DesignPatternNode` indicates that a structure where a specific design pattern could be used has been found. This is specified by the association between class `DesignPatternNode` and `DesignPattern` in Figure 74. The `htmlFile` attribute is used to reference an html file describing the design pattern and how it can be applied to the candidate structures identified by the decision tree. The `Role` class, also shown in Figure 74, is used to highlight to the user key UML elements for the application of the design pattern. It is expected that the roles would be discussed in the provided overview of the design pattern. The `RoleInstance` class relates the `Role` class to the `NodeParameters` produced by the decision tree. When a structure has been found, this tells the user which roles are played by the model elements in the structure.

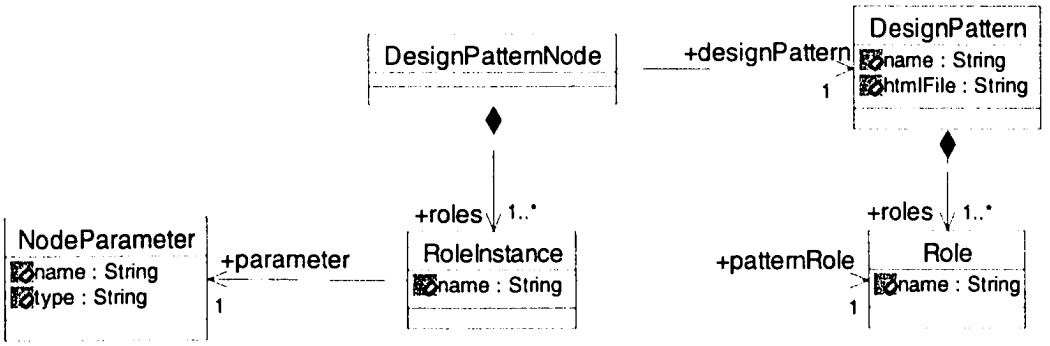

Figure 74 - DesignPattern and DesignPatternNode Relationship

This decomposition between the (static) description of a pattern and its link to the node in the decision tree (i.e., classes `DesignPatternNode`, `DesignPattern`, and `Role`) on the one hand, and the (dynamic) identification of structures in the model (i.e., classes `RoleInstance` and `NodeParameter`) on the other hand, allows several model structures

to be matched to a given design pattern. This is illustrated in Figure 75 where two structures in a model are found to be candidates for the application of the *State* design pattern.

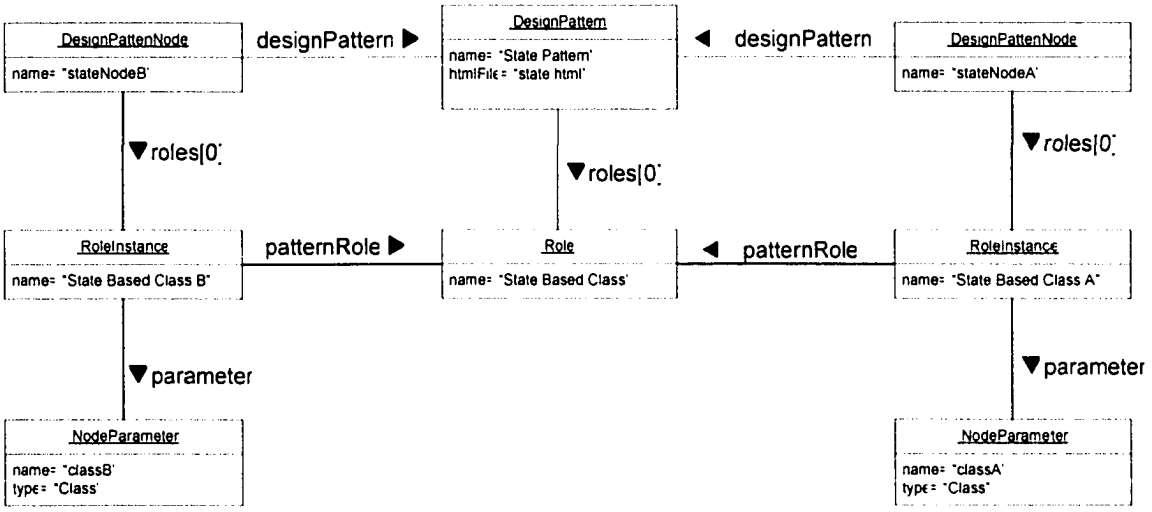

Figure 75 - Instantiation of DesignPattern and DesignPatternNode

9.2.5 DPA Model Instantiation

In the previous section specific sample instantiations were provided to aid the reader in understanding the `DPA Model`. This section elaborates on this approach by providing a full working instantiation of the `DPA Model`. The decision tree presented in Figure 76 is a simple decision tree similar to those presented in previous sections. The purpose of this decision tree is to identify abstract classes which can easily be replaced by an interface. This is done by identifying an abstract class with no implemented methods and identifying the class which derive from this class. Figure 77 illustrates the instantiation of this decision tree using the DPA metamodel.

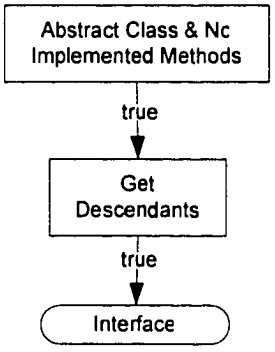

Figure 76 - Sample Decision Tree

A brief description of the nodes is given below.

- *Abstract Class & No Implemented Methods:* Check if the class is abstract, while not defining any concrete operations.
    - *in class:Class* - the class being investigated.
- *Get Descendants:* Retrieve the descendants of this class.
    - *in class:Class* - the class being investigated.
    - *out descendants:Set(Class)* – the descendants of the class.

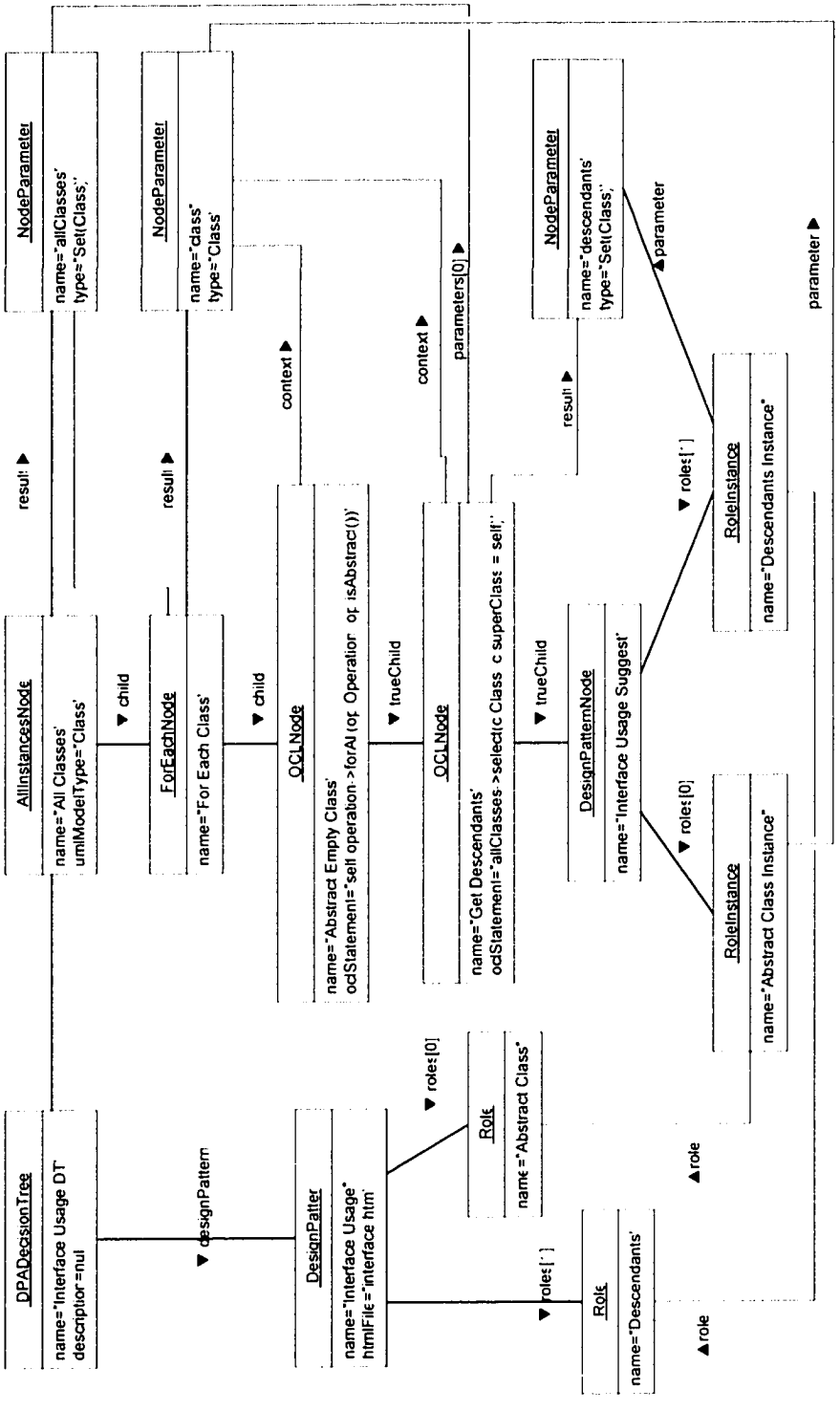

Figure 77 - DPA Model Instantiation

Instantiating the DPA metamodel is automated by virtue of using EMF. As mentioned previously EMF can generate a basic editor for the models generated by EMF. The use of this editor will be illustrated using this instantiation example. The EMF editor is an Eclipse plugin and once integrated into Eclipse the DPA model can be generated under existing projects (Figure 78). The DPA instance is created with the `DecisionTree` object as the root of the tree (also shown in Figure 78). The *Properties* view allows the developer to fill in the attributes of the object.

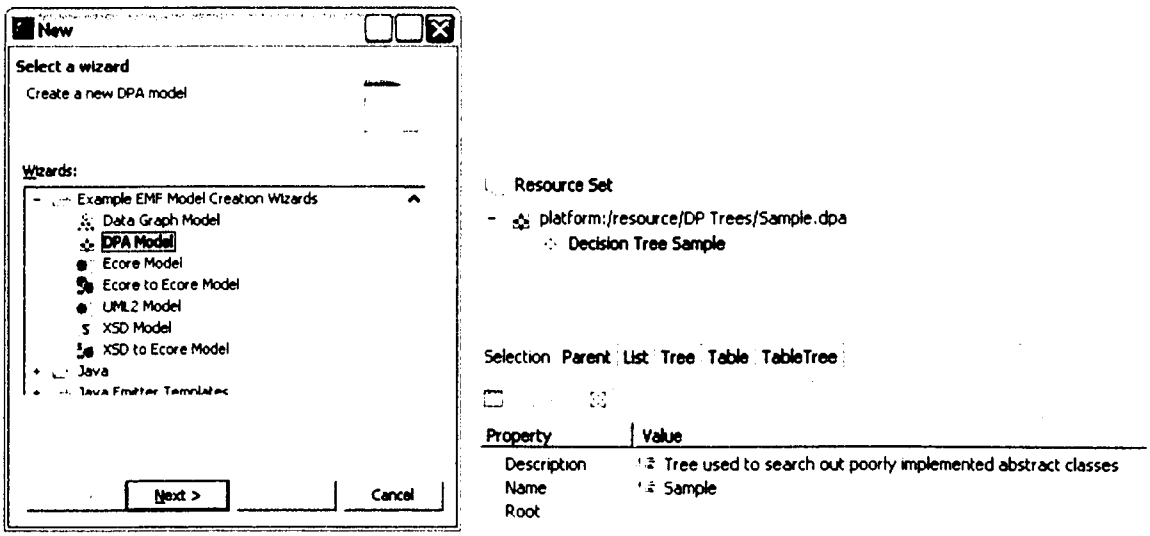

Figure 78 - Creating a new DPA model

The next step is to populate the decision tree with the required nodes and design patterns (Figure 79). Once all the nodes are created the NodeParameters can be created in the same fashion as the nodes themselves (Figure 79). Once all the required objects for the DPA model are created the information for each node (attributes) can be edited in the *Properties* view, this includes the transitions (associations) between nodes and the use of parameters (Figure 80).

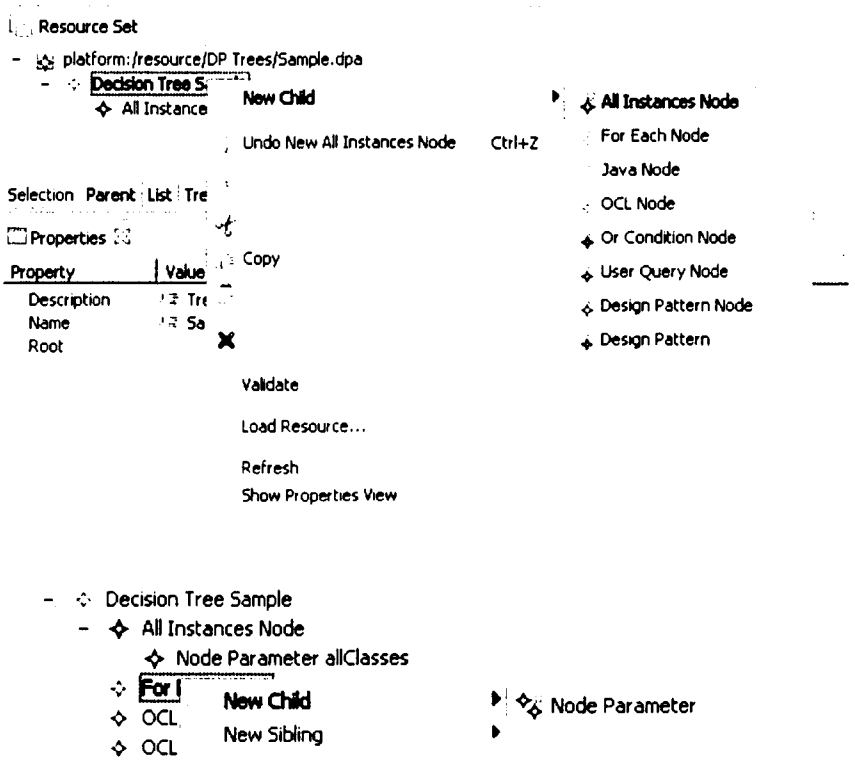

Figure 79 - Creating New Objects Using the DPA Editor

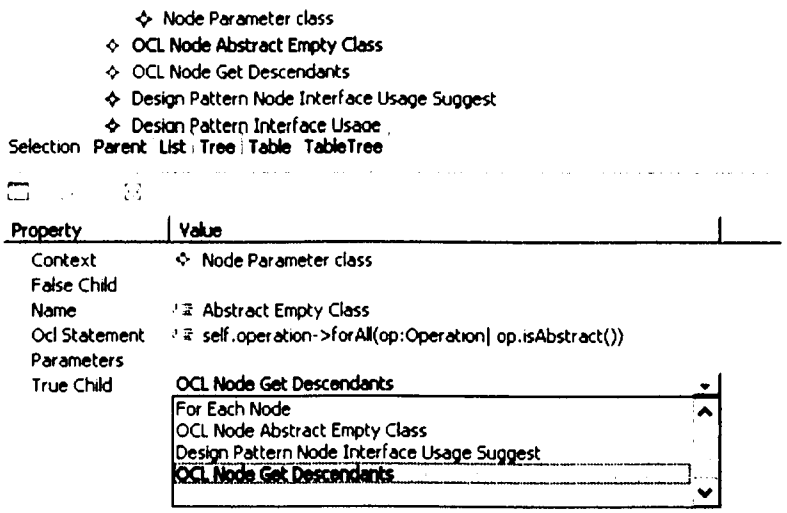

Figure 80 - Editing Attributes and Associations using DPA Editor

This is a simple example where no Java classes or HTML files needed to be created. Obviously for a full decision tree these files will have to be created and referenced by the DPA model. See the following section for further details.

9.2.6 Required Expertise

In order to design and develop the design trees discussed in this thesis, the designer requires specific knowledge:

1. The designer of the decision trees must understand the design pattern for which the decision tree is being developed against. To understand the design pattern's consequences, its benefits, general structure, roles and relationships between the acting components. This knowledge is required to define effective candidate structure on which the design pattern can be applied.

2. The designer must have a strong grasp of the UML 2 meta-model. This is essential as the designer must be aware of where the needed information is present in the UML model and the structure of the model in order to extract the information.

3. Because the decision trees are first specified in OCL and OCL can be used in the implementation of the decision trees, the decision tree designer must be familiar with using OCL language.

4. The designer must have a strong knowledge of Java. This is required to develop the analysis where OCL is not sufficient and the flexibility of Java is required. The knowledge of EMF is an asset as the designer can leverage some of the benefits of this framework mentioned in section 9.1.1.

5. Knowledge of HTML can be also an asset as HTML pages are used to interact with the user. This knowledge is not considered required however as there are several commercial tools available for the authoring of HTML pages [35].

Based on the strong knowledge requirements and expertise, it is assumed that the designer of the decision tree will be a senior developer. The time required to develop a decision tree will be proportional to the designer's knowledge of the above mentioned criteria, as well as the complexity of the candidate structure.

The knowledge required by the end user will be simply a strong grasp on object-oriented concept, a strong grasp of UML and the ability to apply the suggestions proposed.

9.3 The DPA Processing Engine

The DPA Processing Engine is responsible for performing the actual analysis of a given model, according to a specific decision tree. Figure 81 shows the main classes that make up the processing engine: Classes implementing interface `NodeProcessor`, and in charge of actually processing each kind of node in the decision tree are not shown. The `DTProcessor` class is responsible for performing the analysis of a given model according to a specific decision tree.

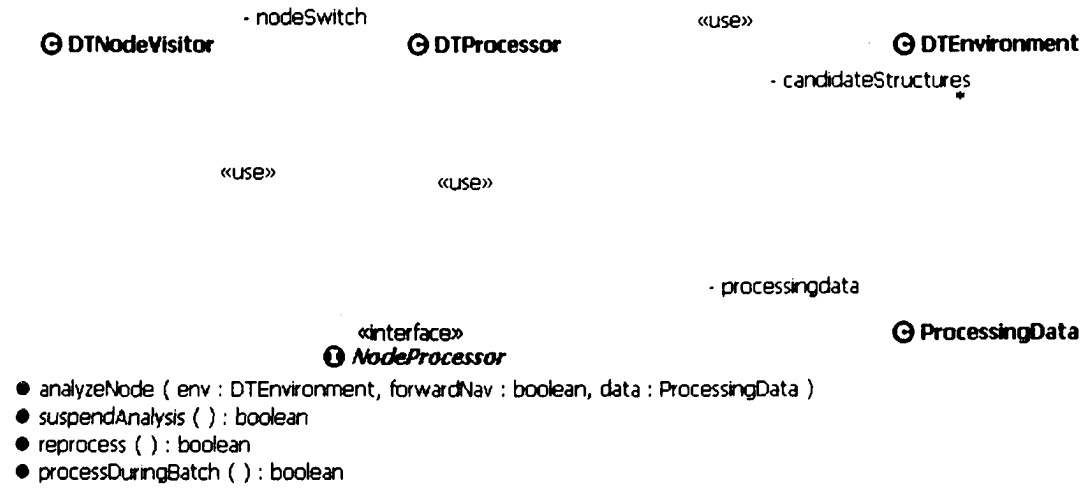

Figure 81 - DPA Processing Engine Main Classes

In order to process the individual nodes on the tree, an adapted *Visitor*/Factory pattern was used. For each type of node in the `DPA Model` there exists a class implementing the `NodeProcessor` interface and able to process the node (hereon referred to as a node

processor). When a node has to be processed, `DTProcessor` asks `DTNodeVisitor` for the adequate node processor (only `DTNodeVisitor` knows the mapping between the nodes and the node processors). The `analyzeNode()` method can then be invoked by `DTProcessor` on the returned node processor object. This is illustrated, abstracting out low level implementation details, in Figure 82 (explicit message names instead of calls to actual operations in the implementation are used). In the main loop, `DTProcessor` asks get the next node to be processed from `DTEnvironment`. It then asks `DTNodeVisitor` for the processor for that type of node, and then asks that processor to analyze the node. The `analyzeNode()` operation typically gets the node to be analyzed from `DTEnvironment`, gets information about the node and uses/creates instances of `NodeParameter`. This process (i.e., `DTProcessor` asking `DTNodeVisitor` for a `NodeProcessor` instance) starts with the root node in the decision tree (recall Figure 69). It ends when `DTEnvironment` does not return any new node to process. The use of this pattern facilitates future extensions of the processing engine: if a new node (and corresponding processor) is required (i.e., there is a change to the `PDA model`), `DTNodeVisitor` has to be updated, but `DTProcessor` does not.

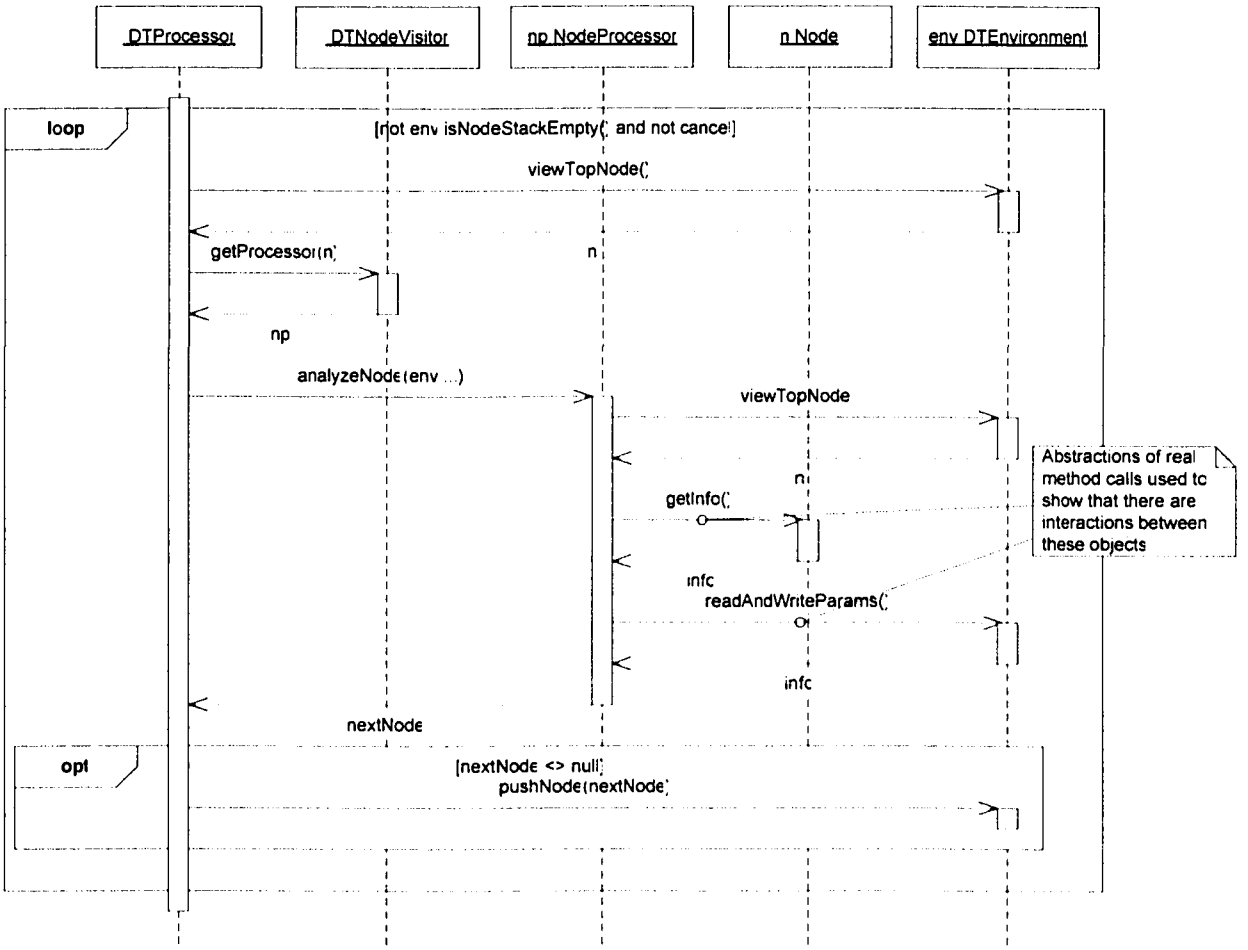

Figure 82 – DPA Processing explained

As can be seen from Figure 81, the `analyzeNode()` operation accepts a `DTEnvironment` object and a `ProcessingData` object as parameters. These objects are used by the node processors to get and pass information. The need for two separate objects is based on the type of information that is used: The `ProcessingData` class is used as an entity class containing information that is application dependent (e.g., the instance of the UML2 metamodel), while the `DTEnvironment` class contains information that is decision tree specific (e.g., the decision tree nodes).

The `ProcessingData` class contains the model being analyzed and the GUI interface. It also contains information on the identified candidate structures, and the directory containing information needed by the node processors (e.g., location where to find Java

classes for `JavaNode` and `JavaDirective` objects). The directory uses a special structure containing several subdirectories to store related files. The HTML subdirectory contains html files used by the `UserQueryNode` processor and the `DesionTreeNode` processor to display information to the user. The html files are the simplest and most versatile way of presenting information to the user. Note that the `DesignPatternNode` and `UserQueryNode` classes have references to these html files with the `htmlFile` attribute.

The JAVA subdirectory contains the compiled byte code files for the classes to be dynamically created and used by the `JavaNode` processor. As can be seen from Figure 83, the `JavaNode` contains only information need to identify the class to instantiate. Once this class is instantiated using dynamic loading (a customized `ClassLoader` was created for this purpose) the `analyze` method is invoked to perform the required analysis on the UML model. The `JavaDirectives` are used as arguments to allow the class to change its operation for different decision trees (see section 9.2.2). As always the result from the analysis guides the DPATool to the next node to be evaluated.

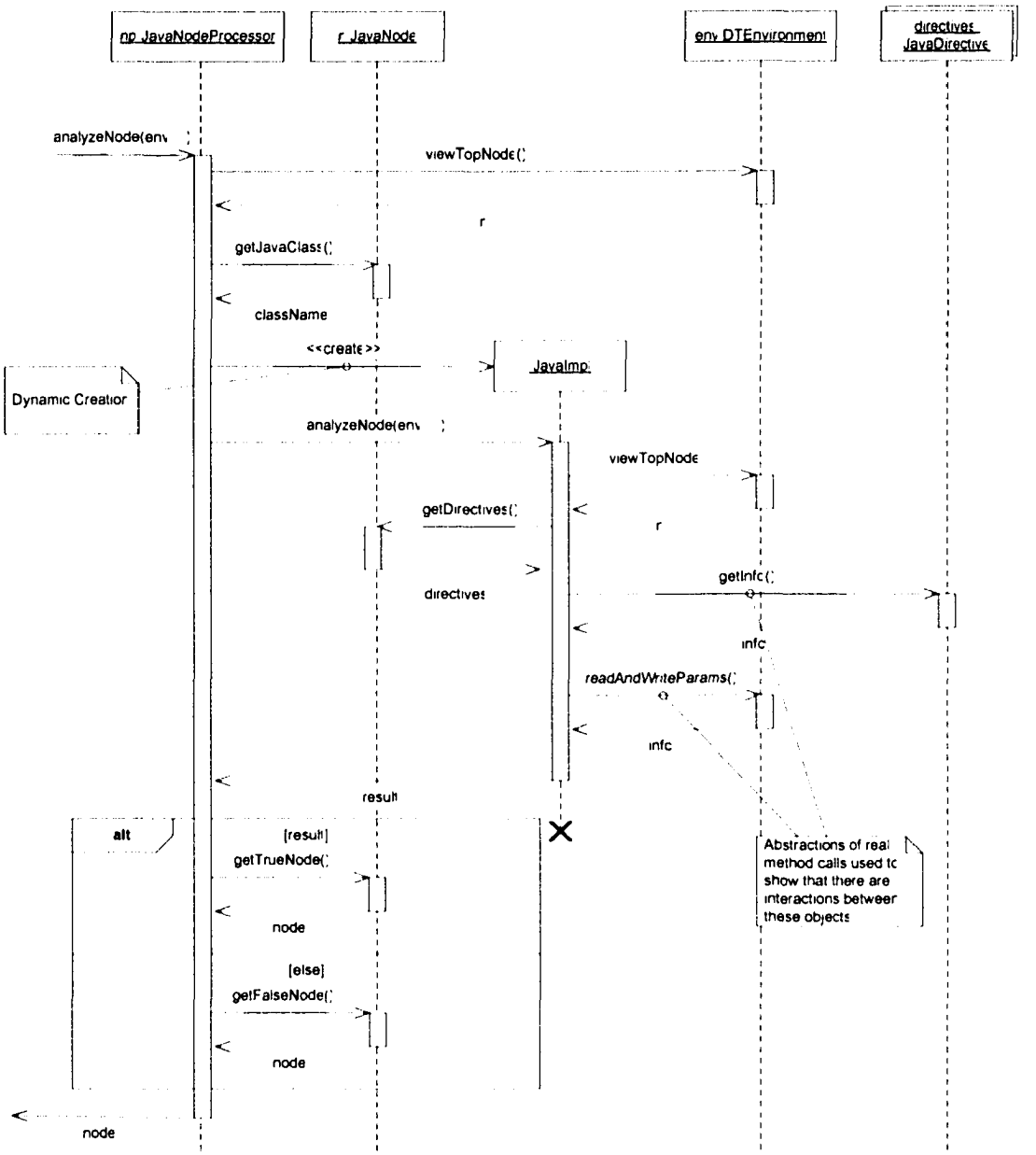

Figure 83 - JavaNode Processing

The EXT subdirectory contains compiled byte code for the model extensions to be used by the OCL Engine (see section 9.1.2.2).

The `DTEnvironment` class contains `Node` instances organized into a stack, from which `DTProcessor` pops elements to be processed and to which `DTProcessor` pushes the next elements to be processed. `DTEnvironment` also contains the decision tree being used for

processing, and the shared hash used by the node processors to manipulate `NodeParameter` instances. The `DTEnvironment` objects also act as a result returned by the `DPA Processing Engine`, as all the needed information regarding the candidate structures can be extracted from these objects.

The `DPA Processing Engine` can work in interactive mode or in batch mode. If in batch mode and the user input is required, the analysis is suspended: operation `suspendAnalysis()` in interface `NodeProcessor`. The processing engine can however continue executing the part of the analysis that is not dependent on the user input. This allows the DPATool to perform as much processing as possible in the absence of the user. Once the user is available the suspended analysis will resume by gathering the needed information. Operation `processDuringBatch()` in interface `NodeProcessor` tells the `DTProcessor` object whether the node can be processed in batch mode or not (in which case the analysis is suspended): e.g., the node processor processing a `UserQueryNode` instance returns false.

The `DPA Processing Engine` has an additional functionality that can be used to speed the analysis, referred to as the *quick analysis* feature. The quick analysis feature allows the decision tree designer to specify, for a `DesignPatternNode` instance, a node above in the decision tree to be evaluated immediately after finding a candidate structure: this is the purpose of association between class `DesignPatternNode` and class `Node` with rolename `resetNode` (Figure 70). This was introduced to avoid processing every path of an `ORConditionNode` once a design pattern can be suggested in one of the "or" branches of the tree[8]. For instance, consider to the decision tree in Figure 43 (which is duplicated in Figure 84 to ease our discussion) for the identification of candidate structure for the *Observer* pattern. Assuming the `DesignPatternNode` instance corresponding to the "Observer Pattern" node in the tree has a link to node "Observer Method Call", and

---

[8] This feature is similar to the evaluation of disjunctions in some C compilers: i.e., when a condition is made of two disjuncts (e.g., `a==1 || b>10`). Then, if the first term of the disjunct evaluates to true, there is no need to evaluate the second disjunct.

assuming the quick analysis feature is on, it is only necessary to visit the `DesignPatternNode` instance once for one of the "or" branches (instead of three times) to give a result to the user. If the quick analysis feature is disabled or a `resetNode` is not specified, then the `DTProcessor` simply analyses all the "or" branches.

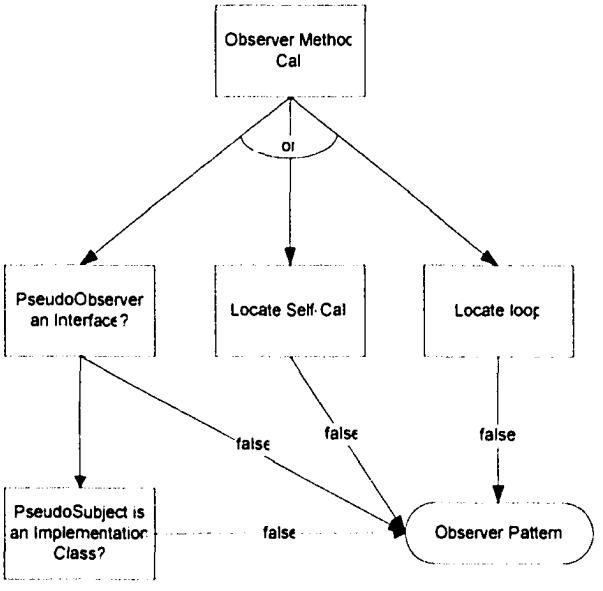

Figure 84 - *Observer* Pattern Decision Tree (duplicate of Figure 43)

9.4 The DPA GUI

The `DPA Processing Engine` was designed to be used as a library component and can be integrated into a variety of case tools. However it needs to interact with the user to retrieve information and update the user on the progress of the analysis. To have such an interaction as flexible as possible, a set of interfaces were developed which allow the `DPA Processing Engine` to work with any GUI implementation compliant to those interfaces. Figure 85 provides a class diagram with those `DPA GUI` interfaces.

Figure 85 - DPA GUI Overview

The `DPA_GUI` interface is the main interface of the set. It provides a means to access the other objects implementing the `UserQueryPage` and `ProcessingPage` interfaces, through the `getUserQueryPage()` and `getProcessingPage()` operations. The `setDPAPage()` operation is used to indicate what information should be displayed to the user. The `DPA_GUI` interface also provides several operations for the batch mode operation of the DPATool. The `DPA Processing Engine` may enter batch mode after a prolonged period of user inactivity through the `setBatchMode()` operation; while operation `batchMode()` is a query operation used to determine if the user has enabled batch mode. Operation `waitOnBatchMode()` is provided as a means to block the processing thread until the `batchMode` is disabled by the user. Operation `requestProcessingExit()` is a query operation used by the `DPA Processing Engine` to determine if the user has cancelled the analysis. Finally the `displayResults()` operation is used to indicate that processing is complete and the `DTEnvironment` objects for the candidate structures found are provided in the parameters.

The `UserQueryPage` interface is used by `UserQueryNode` to display html files containing the question to be asked of the user. A screenshot of the current implementation is shown in Figure 86. It relates model elements found in the model under analysis (i.e., the "The combined fragment ATMv5TestDriver ..." part of the message) to the question being asked ("Is the combined fragment ..."). (Note that the model elements referenced in the html file are obtained from the hash in the DTEnvironment.) The user's response is then available through the `getUserResponse()` operation.

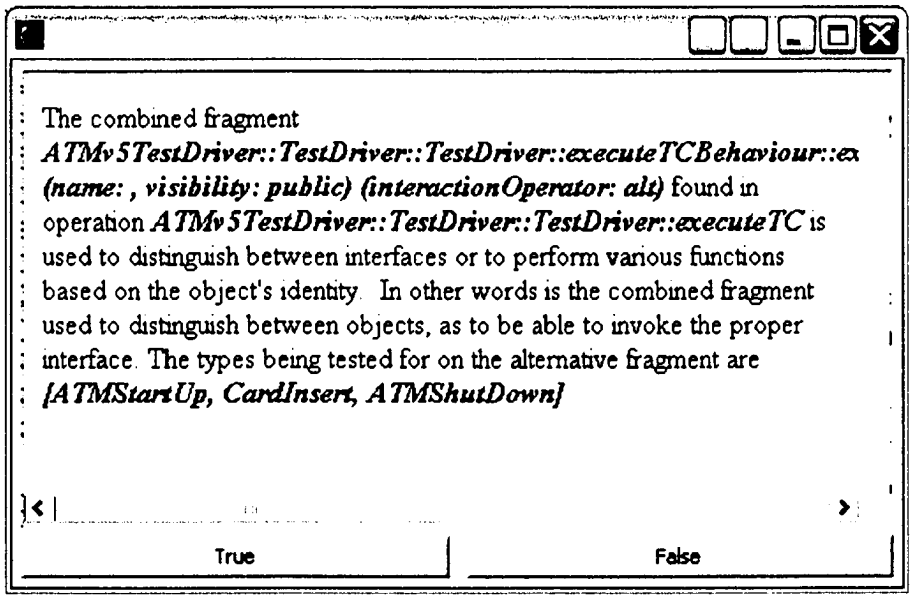

Figure 86 - User Query Window

The `ProcessingPage` is optional and used to display information regarding the analysis, such as what decision tree is being used presently, which node is being processed, and if batch mode is enabled.

As mentioned previously, the `DTEnvironment` objects are used as results from the `DPA Processing Engine`. The page to display the results, although not technically part of the `DPA GUI` module is discussed here as it must make use of the `DTEnvironment` objects provides. Therefore it somewhat interacts with the `DPA Processing Engine`. The result page contains two panes (Figure 87). The first one, on the left reports, under the form of a tree structure, on all the design patterns found when analyzing the model. The top level

nodes of the tree are the design patterns. For each design patterns, expanding the node reveals the different roles involved in the pattern and the model elements playing those roles (Figure 87). Under each of the model elements there are the `DesignPatternNodes` (leafs of the decision tree) where the model element was found in that role. (When a model element contains many `DesignPatternNodes` this suggests stronger evidence that the model element is playing that role in the UML model as it was found several times.) When a node representing a design pattern is selected, the second pane (right pane) displays the html file describing the pattern and listed in the `DecisionTree` class (Figure 69). This right pane is also used to display the `DTEnvironment` information, as shown in Figure 88, when a node corresponding design pattern node is selected (the information displayed in the state of the `DTEnvironment` when the `DesignPatternNode` was evaluated).

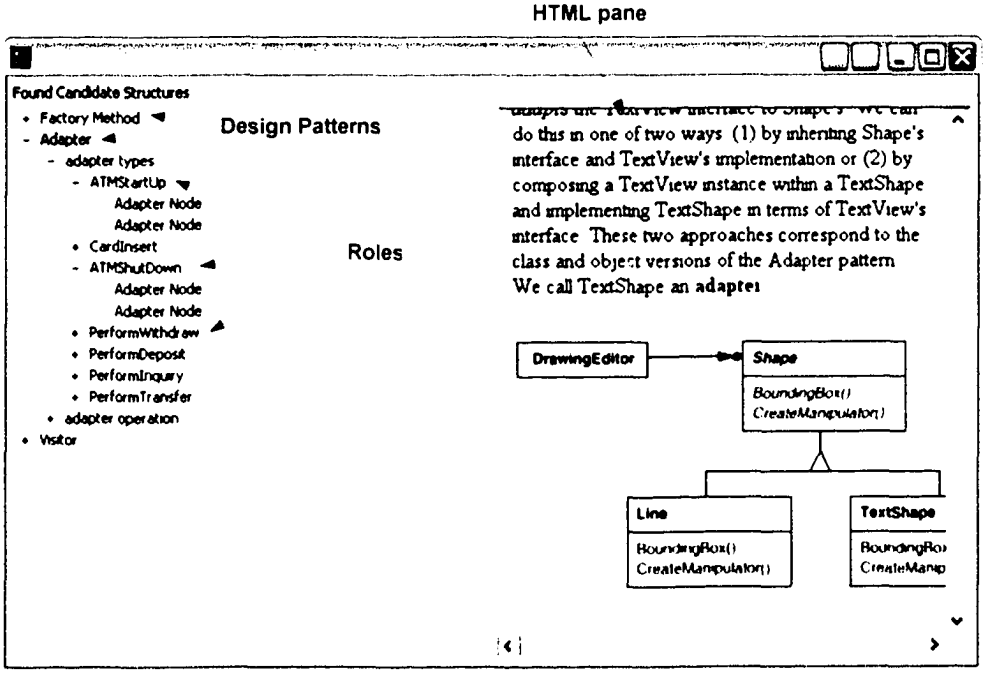

Figure 87 - Results Window (HTML content from [26])

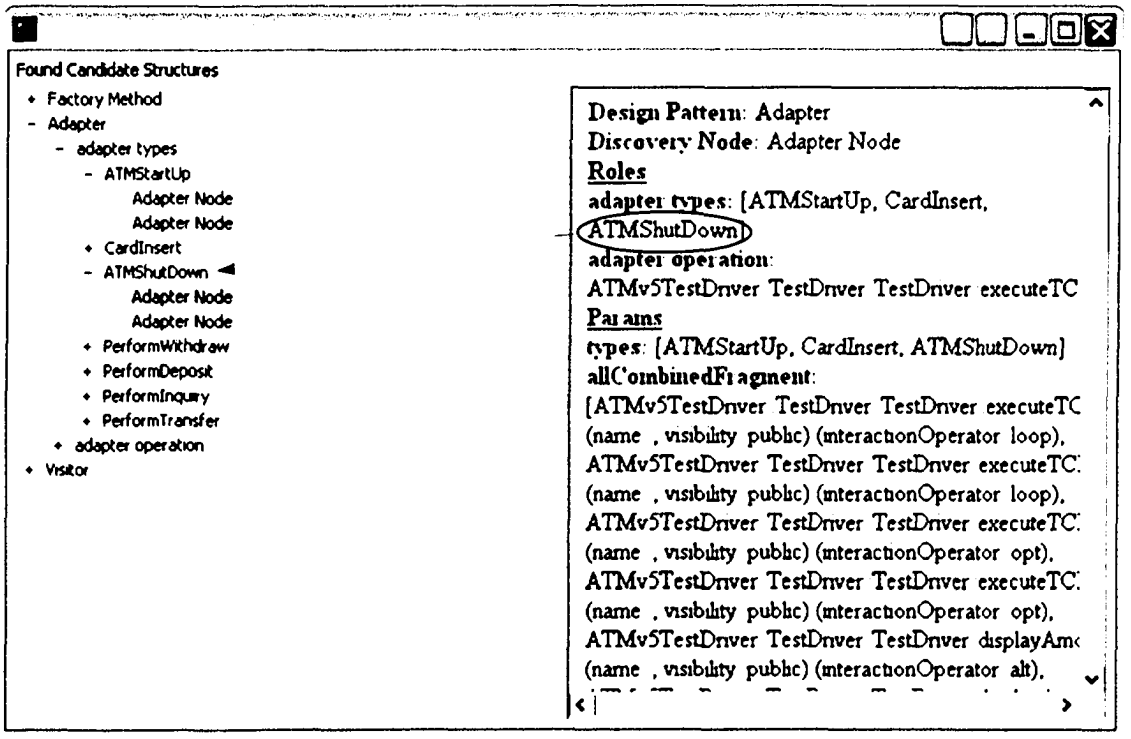

Figure 88 - Results Window Showing DTEnvironment Information

10 CASE STUDIES

We used two case studies to validate our methodology, an Elevator Control System [33] and an ATM Test Driver [8]. Both these software systems were developed by fourth year students at Carleton University. The students were familiar with modern analysis and design methodologies using UML, and had extensive training in the Java programming language through their courses. The students have also been introduced to the GoF patterns in a 4th year software engineering course. We will first explain how the UML models were extracted and integrated into our tool and then show, for each case study, what candidate structures were detected and whether they corresponded to real opportunities for using a design pattern.

10.1 Extracting Eclipse UML2 Models

Both systems had accompanying UML documentation, however these were generated using the Together case tool [5]. The UML model generated by the Together case tool is a proprietary UML 1.4 model, therefore the Eclipse UML2 models (referred hereon as the UML2 models) to be used by the DPATool needed to be generated for both projects.

There were several steps to generating the UML2 models for these systems.

1. The static components of the UML2 models were generated from the Java source code using the Model Transformation Framework (MTF) [31] developed by IBM. The MTF works in conjunction with the Eclipse Java Development Tools (JDT) [19] and is therefore limited to the static components of the UML2 models. A static UML2 model can be also generated from source code using the Rational Software Architect (RSA) [44] tool. However there were inconsistencies in the model generated by the RSA tool in regards to using classes found in the Java APIs [46].

2. The RSA tool was used to generate the dynamic portions of the UML2 models, as it provides the functionality to visualize a method on a class as a UML2 sequence diagram. The visualization (also referred to as a 'Topic Diagram' in the RSA tool) is dynamic, and will reflect the changes made in the Java method. The Topic diagram can be persisted as a 'Diagram File'. To create the Diagram file an accompanying UML2 model describing the sequence diagram is generated. The root of this model is an instance of the UML2 `Collaboration` representing this method which can be extracted and integrated into the static UML2 model generated in the first step.

3. Small modifications were then performed on the model by an automated script which matched the `Message` elements in the dynamic portion of the model to the proper `Operations` within the static portion of the model.

Therefore the UML2 models used in the case studies were generated mainly through third party tools. These models approximate the UML2 models that will be generated by case tools, such as RSA, once that functionality becomes available in future versions. Due to the time required to generate the collaboration elements of the UML2 models, these were only generated for specific methods in each system. The methods were chosen based on the presence of coding constructs found in the methods that should be replaced with design pattern implementations. In all there are 9 sequence diagrams generated for the ATM Test Driver case study and 7 sequence diagrams generated for the Elevator Control System case study.

The time for the processing of the models for the case study is given in the table below. Note that the time shown here is processing time only, not time awaiting a user response.

Table 3 - DPATool Processing Time

Project	Time for Processing
ATM Test Driver	5489 msecs.
Elevator Control System	2342 msecs.

10.2 Elevator Control System

The purpose of the Elevator Control System (ECS) project was to design and implement the Elevator Control System as described in [33], using UML-based OO design. The ECS simulates the control of multiple elevators, in a real-time environment, by scheduling elevators based on requests from the users at various floors. The ECS uses interface classes to emulate the IO hardware that would be present in a real world application.

The ECS system has not been decomposed into subsystems within the implementation; however there is a clear set of classes used to model the problem (`Elevator`, `Door`, `Floor`, etc) and a set of classes used to model interfaces to IO devices (`ElevatorButtonInterface`, `FloorLampInterface`, `OverridePanelInterface`, etc). A third set of control classes were also developed (`ElevatorControl`, `FloorControl`, etc); however the concept of control classes as described in [9] was not properly applied in the ECS implementation and these classes provide very little added functionality. A complete class diagram for this system can be found in Appendix I and some simple metrics for the system are in Table 4.

Table 4 - ECS Metrics

Metric	Value
Number of Classes	29
Number of Methods	159
Number of Attributes	106
Number of Use Cases Implemented	10

10.2.1 State Pattern

The control of each elevator within the ECS system is performed by an `Elevator` object. The elevators are state based with several states: idle, moving, etc. as described in [33]; thus making the `Elevator` class an ideal candidate for the *State* pattern. However the current implementation does not make use of the *State* pattern, rather a set of attributes is

used to describe the state of the object and these attributes govern behaviour of the object. Figure 89 illustrates a portion of the sequence diagram generated for the `moveElevator` method. The method calls between objects have been omitted for simplicity. The source code for this method can be found in Appendix J. The occurrences where an attribute was used in the guard condition are highlighted in the sequence diagram. The loop structures in the diagram represent 'while' statements found in the source code.

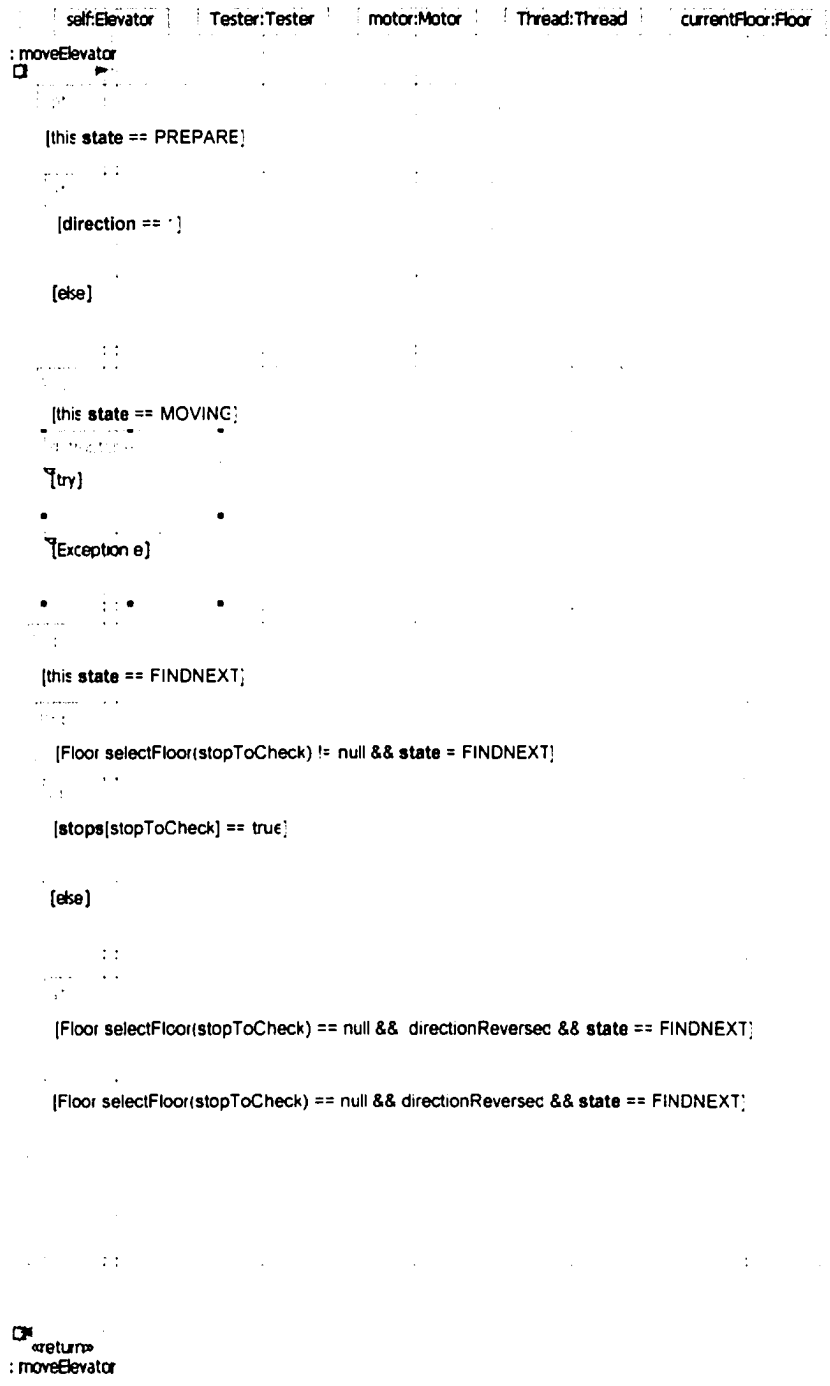

Figure 89 - moveElevator Partial Sequence Diagram

When the ECS system was analyzed by the DPATool, the `Elevator` class was flagged as a potential candidate for the *State* pattern.

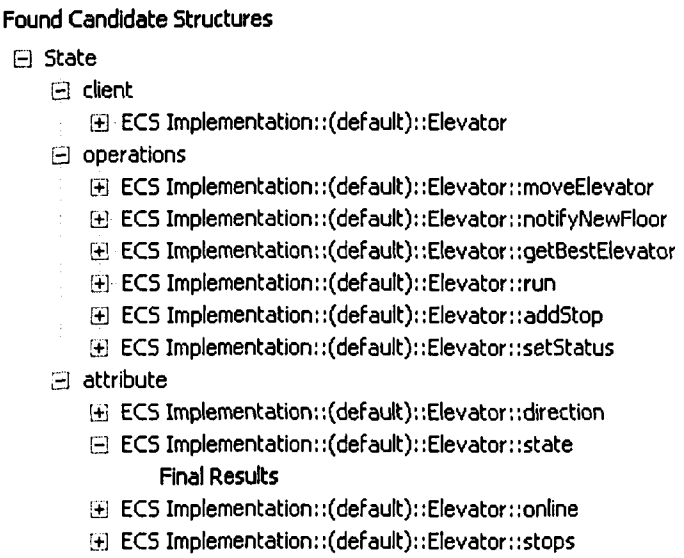

Figure 90 - DPA Results for ECS system

DPA DT Name: State Pattern

Design Pattern: State
Discovery Node: Final Results
Roles
client: ECS Implementation::(default)::Elevator
operations: [ECS Implementation::(default)::Elevator::moveElevator, ECS Implementation::(default)::Elevator::run, ECS Implementation::(default)::Elevator::addStop, ECS Implementation::(default)::Elevator::getBestElevator]
attribute: ECS Implementation::(default)::Elevator::state

Figure 91 - Details for *state* attribute

Figure 90 shows the `Elevator` class, with several attributes used to determine the proper behaviour of the object. The DPA decision tree for the *State* pattern also gathers the operations in which these attributes are being used to control the object's behaviour. The detailed results for the `state` attribute, as seen in Figure 91, indicate that this attribute is used in the `moveElevator`, `run`, `addStop`, and `getBestElevator` methods. A complete description of the attributes used within each method is given in Table 5.

Table 5 - Elevator 'state' attributes and corresponding methods

Attribute	Method
state:int	moveElevator, run, addStop, getBestElevator
Direction:int	moveElevator, notifyNewFloor, getBestElevator
online:boolean	addStop, getBestElevator, setStatus, notifyNewFloor
stops:boolean[]	moveElevator, notifyNewFloor

The use of the *State* pattern is warranted, as the current implementation makes modifications to the existing states or addition of new states difficult. The use of four state attributes has created complex and lengthy control structures throughout the class's methods. Extending the system by using inheritance, as is common practice, is not a practical approach given the current implementation. As well a simple change to a given state may require extensive rework of these conditional statements.

Through the introduction of the *State* pattern the addition of new states or transitions between states is simplified to defining new state classes or subclassing existing ones. The application of the *State* pattern would eliminate lengthy control structures, rendering the code more comprehensive and the state behaviour code would be better compartmentalized in separate classes. These benefits of the *State* pattern will make the system easier to maintain and extend.[26]

From the above explanations, we see that the DPATool was correct in recommending the use of the state pattern for the `Elevator` class.

10.3 ATM Test Driver

The ATM Test Driver was developed as a case study for the Regression Test Selection tool (RTSTool) [8]. The case study called for the development of an RTSTool compatible test driver to test an already existing ATM system. The ATM system was included in the UML2 model generated for this case study. Some software metrics for the ATM test driver and the ATM system itself are found in Table 6.

Table 6 – ATM Test Driver Metrics

Metric	ATM Test Driver Value	ATM Value
Number of Classes	15	26

Number of Methods	114	294
Number of Attributes	45	179
Number of Use Cases Implemented		15

The implementation of the test driver consists of several classes representing the use cases of the ATM system (`PerformWithdraw`, `CardInsert`, `GetPin`, etc), the `TestDriver` class to execute the test cases, and the `TestCase` class to represent the various test cases. A class diagram for the test driver can be found in Appendix H.

10.3.1 Factory Method Pattern

Upon instantiation of the `TestCase` class a text string is parsed to determine the use cases for the `TestDriver` class to execute. This functionality is implemented in the `parseUseCases` method; Figure 92 illustrates a portion of the sequence diagram generated for this method. As mentioned in the note within the sequence diagram the interaction operand is preceded and followed by messages relating to the parsing of the `useCases` string which have been omitted. The `parseUseCases` method contains 76 lines of code, 25 of which are found in the if-then-else statement block. Also an important note is that all the objects created within the conditional statement are casted to the `UseCase` type from which they are all derived.

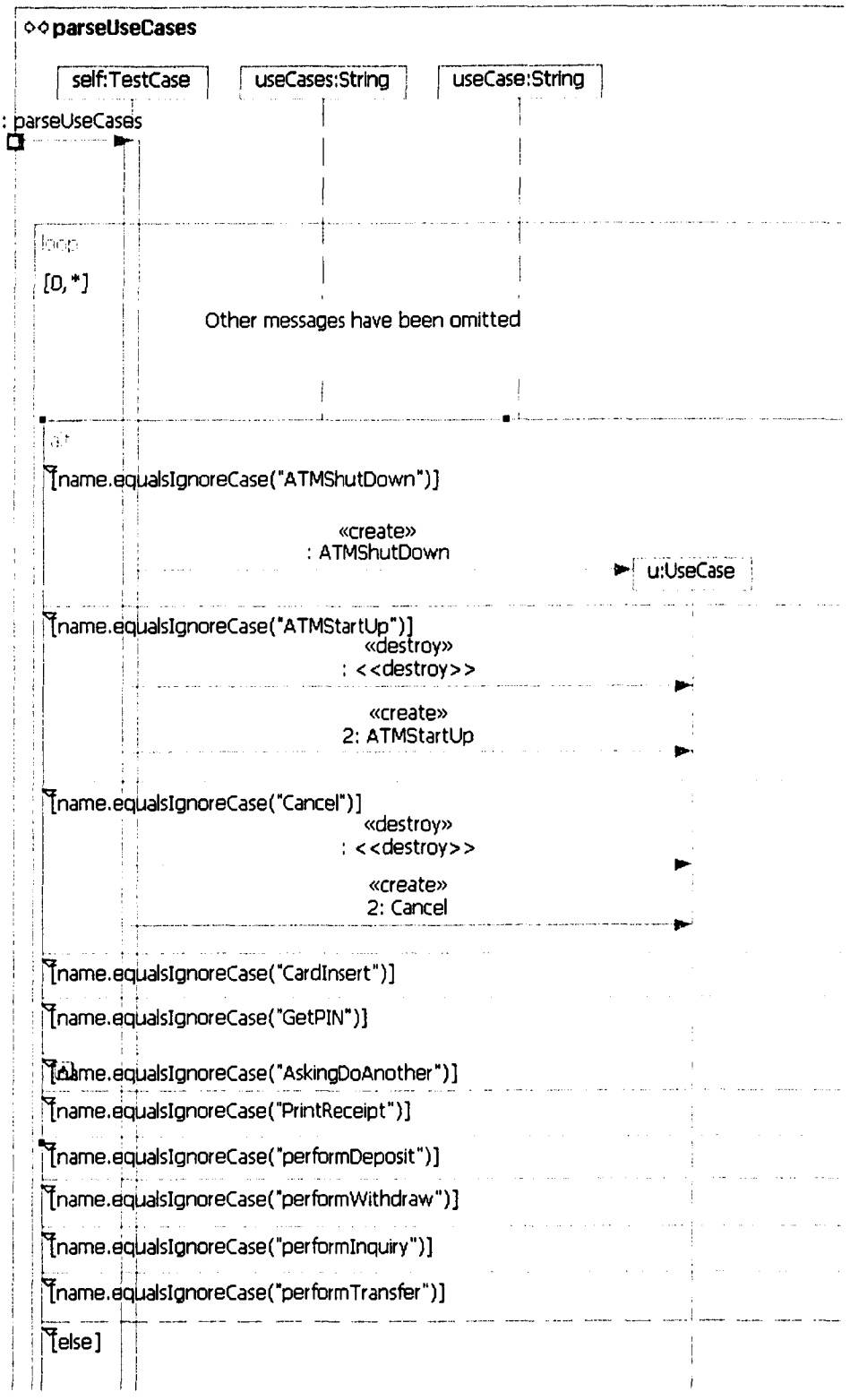

Figure 92 - parseUseCases Partial Sequence Diagram

Upon processing the UML2 model of the system through the DPATool, it was suggested that the *Factory Method* pattern be applied to the conditional statement. The results from the DPATool are shown in Figure 93.

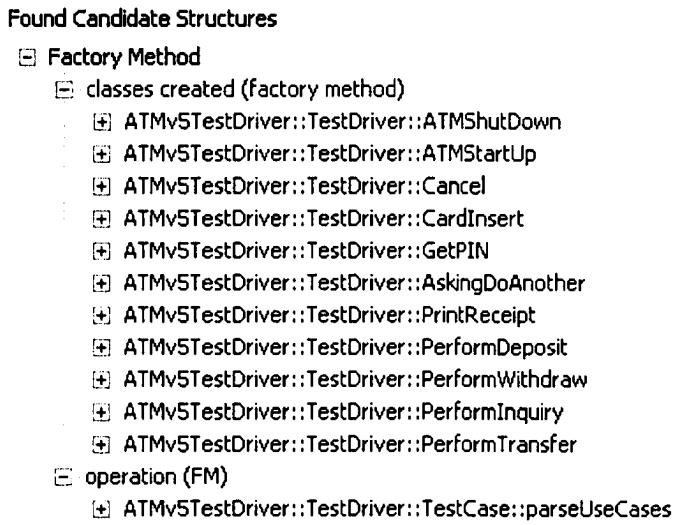

Figure 93 - DPA results for *Factory Method*

This suggestion appears well founded as the current implementation makes it difficult to extend the test driver and add new use cases to the test bed. As mentioned in [26], the *Factory Method* is applicable when "a class can't anticipate the class of objects it must create". Using the current implementation of the system and following the common practice of subclassing to extend functionality, one would subclass `TestCase` and overrides the `parseUseCases` method to add a new subtype of `UseCase` to the system. This would result in having to replicate the 50 lines of code responsible for parsing the text string, which is unrelated to the conditional statement. Applying the *Factory Method* pattern, the conditional statement would be placed in a separate function which would be invoked from the `parseUseCases` method as illustrated in Figure 94. This would also facilitate extending the system by allowing subclasses to extend the `createUseCase` method, as illustrated in Figure 95.

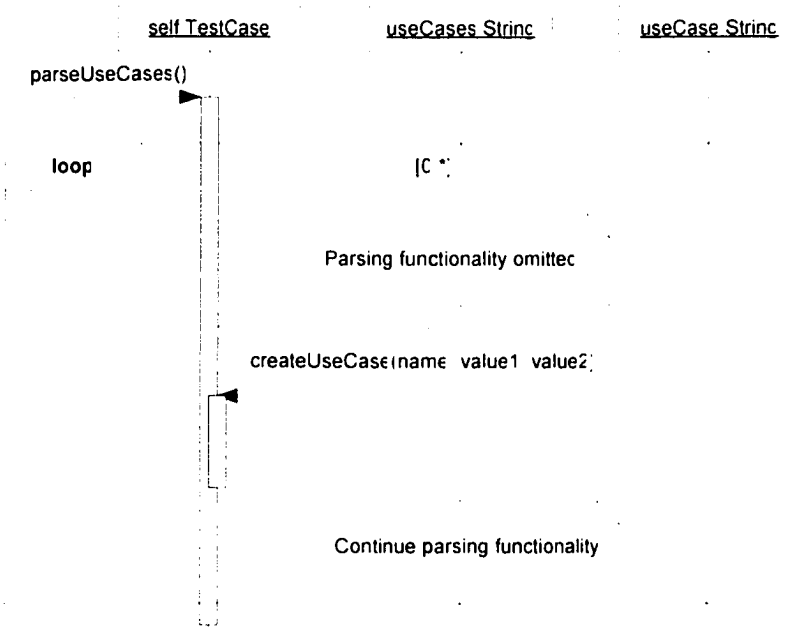

Figure 94 - *Factory Method* applied to parseUseCases

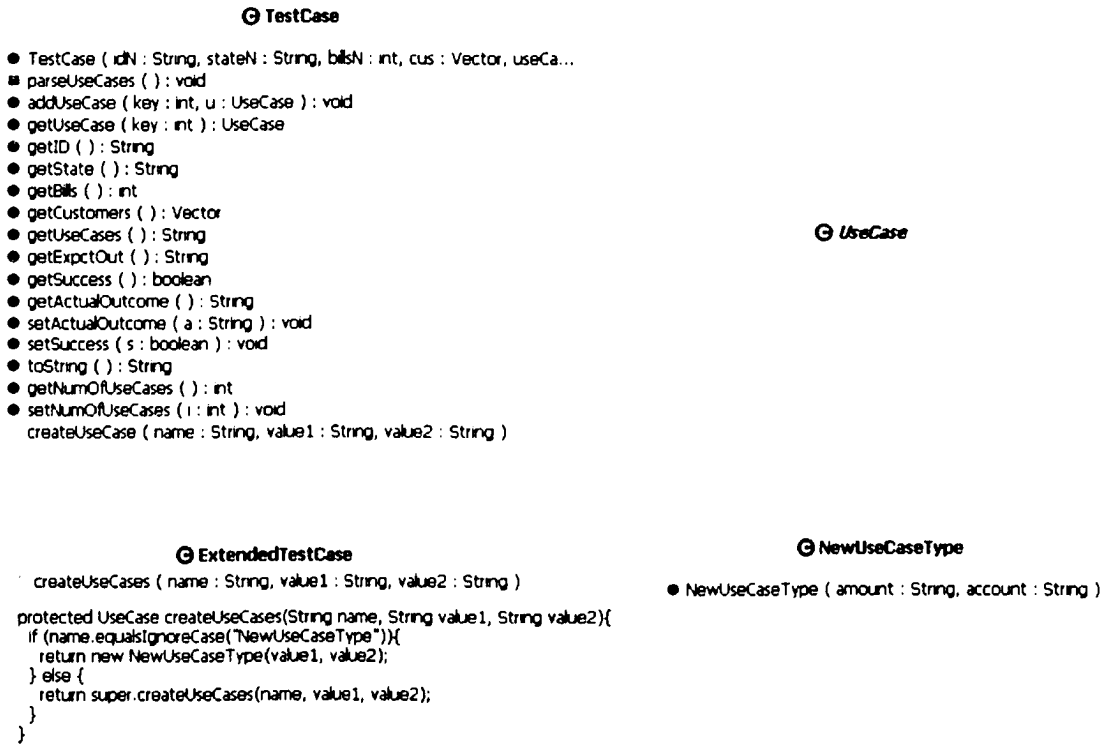

Figure 95 - Extending ATM with *Factory Method* Applied

10.3.2 Visitor Pattern

As mentioned previously, the `TestDriver` class is responsible for executing the use cases provided by the `TestCase` class. Therefore the behaviour the `TestDriver` object depends on which subtype of the `UseCase` class is provided. The current implementation controls the behaviour using conditional statements similar to the candidate structure presented in 7.1.3. A partial sequence diagram (messages are omitted) of an example conditional statement is provided in Figure 96.

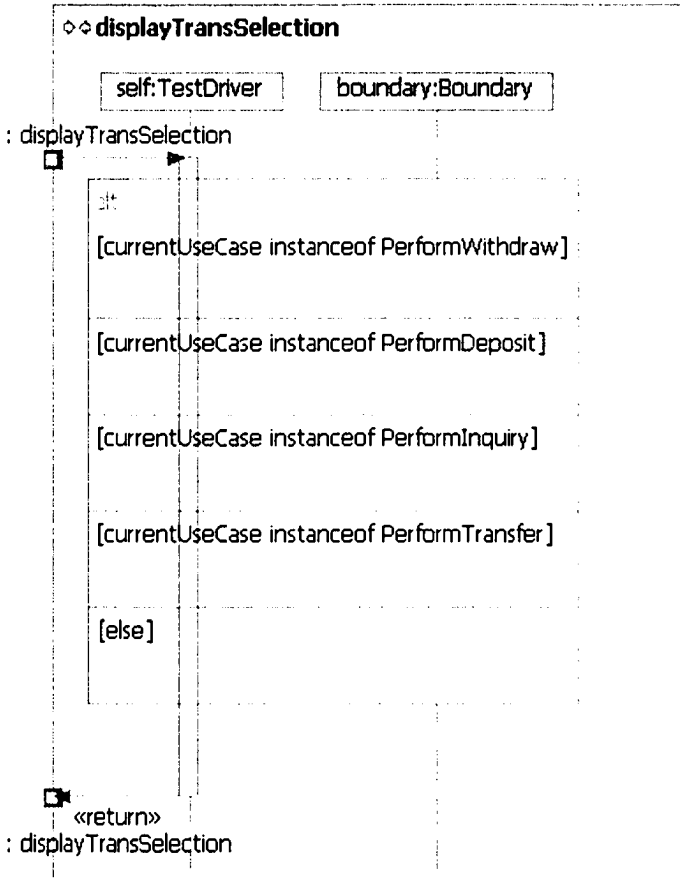

Figure 96 – displayTransSelection Partial Sequence Diagram

This particular conditional statement is repeated in the `requestTransType` method. These conditional statements and one other were flagged by the DPATool as candidate structures for the *Visitor* pattern.

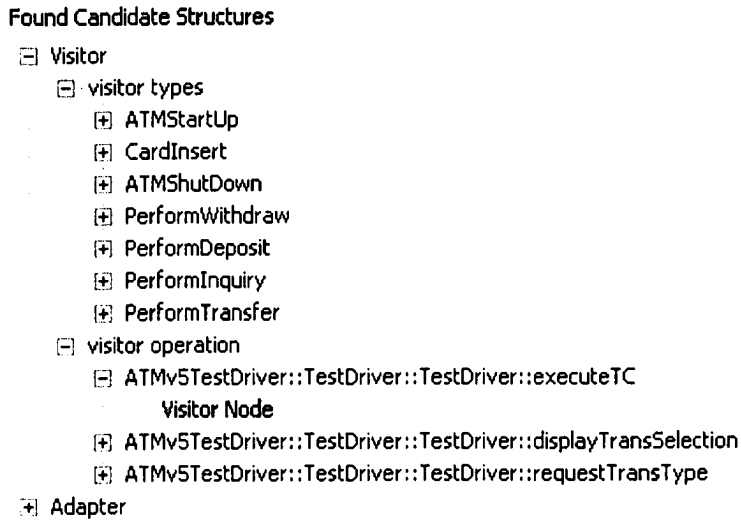

Figure 97 - DPA Results for *Visitor* Pattern

This suggestion again can be fully justified. Adding another subclass of `UseCase` to the system or changing the behaviour of an existing use case could potentially result in the need modify these conditional statements. Again using the common approach of using subclassing to extend functionality, the entire operations would have to re-implemented rather than focusing on the conditional statement. Also the code duplication in the conditional statements found in the `displayTransSelection` and `requestTransType` should be avoided.

Applying the *Visitor* pattern to this set of classes would result in the class diagram on Figure 99. This figure only shows the `ExecuteTCVisitor` class which would replace the source code shown in Figure 98.

```
  public void executeTC(TestCase t) {Figure 98
    … (11 lines of code extracted)
    if (currentUseCase instanceof ATMStartUp) {
      boundary.setAtmState(boundary.ON);
      boundary.setAtmONState(boundary.INITIALIZING);
      boundary.setAtmInitializeState(boundary.GETPASS);
      requestOpPassword();

    } else if (currentUseCase instanceof CardInsert) {
      boundary.setAtmState(boundary.ON);
      boundary.setAtmONState(boundary.WAITING);
      initializeATM();

    } else if (currentUseCase instanceof ATMShutDown) {
      boundary.setAtmState(boundary.OFF);
      requestOpPassword();

    } else {
      //error - a test case sequence may not begin
      //with any other use case
      transactionResults = "Test Case may only start with ATMStartUp,
ATMShutDown or CardInsert";
    }
    … (15 lines of code extracted)
  }
```

Figure 98 – Current executeTC Code Extract

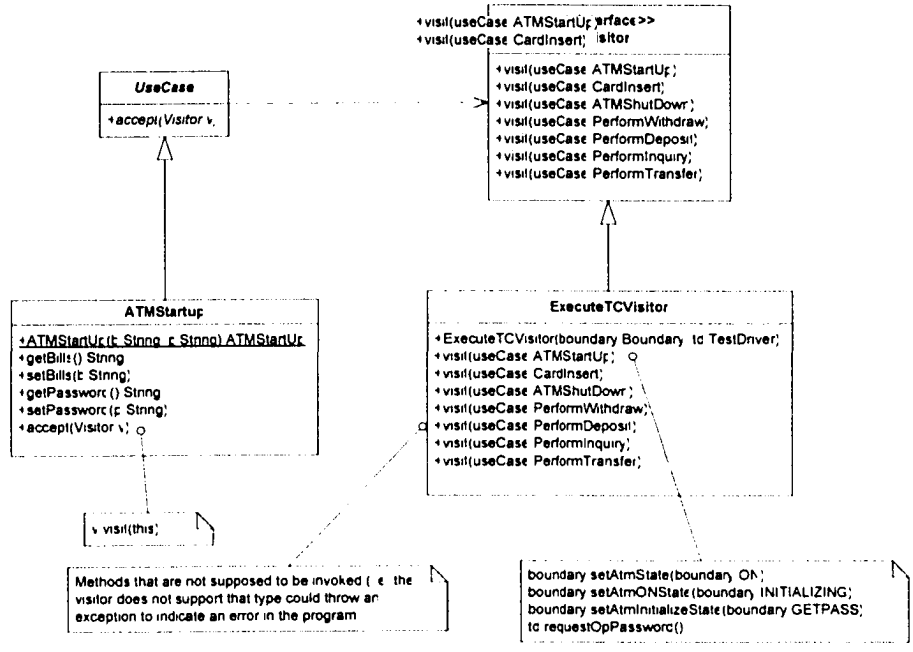

Figure 99 - *Visitor* Pattern Applied to ATM Test Driver

With the pattern applied the executeTC code shown in Figure 98 can be replaced with the code extract below.

```
public void executeTC(TestCase t) {
  … (11 lines of code extracted)
  currentUseCase.visit(new ExecuteTCVisitor(boundary, this));
  … (15 lines of code extracted)
}
```

Figure 100 – Updated executeTC Code Extract

To change the behaviour of the existing use cases, it is simply a matter of overriding the appropriate method in a subclass to `ExecuteTCVisitor`. To add a new use case to the ATM Test Driver will require a little more work, however with the design pattern in place the principles of open-closed may be followed. As an example, a use case is to be added to the system to reset the ATM.

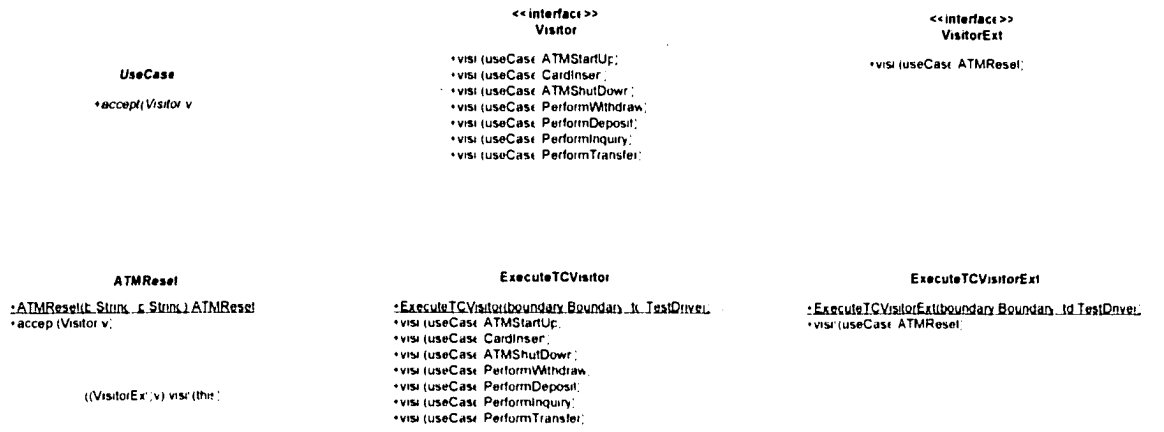

Figure 101 - Use Case Added Using *Visitor* Pattern

By using inheritance, none of the original programming is altered; therefore reducing testing efforts and occurrences of faults in the program. To instantiate the proper visitor object, `ExecuteTCVisitorExt` rather than `ExecuteTCVisitor`, the *Factory Method* pattern can be used.

10.3.3 Adapter Pattern

The `TestDriver` class also contains conditional statements used to determine the object's type to invoke an operation of that object. An example of this type of conditional statement is given in sequence diagram found in Figure 102. These conditional statements were flagged by the DPATool as being candidate structures for the *Adapter* pattern, as show in Figure 103.

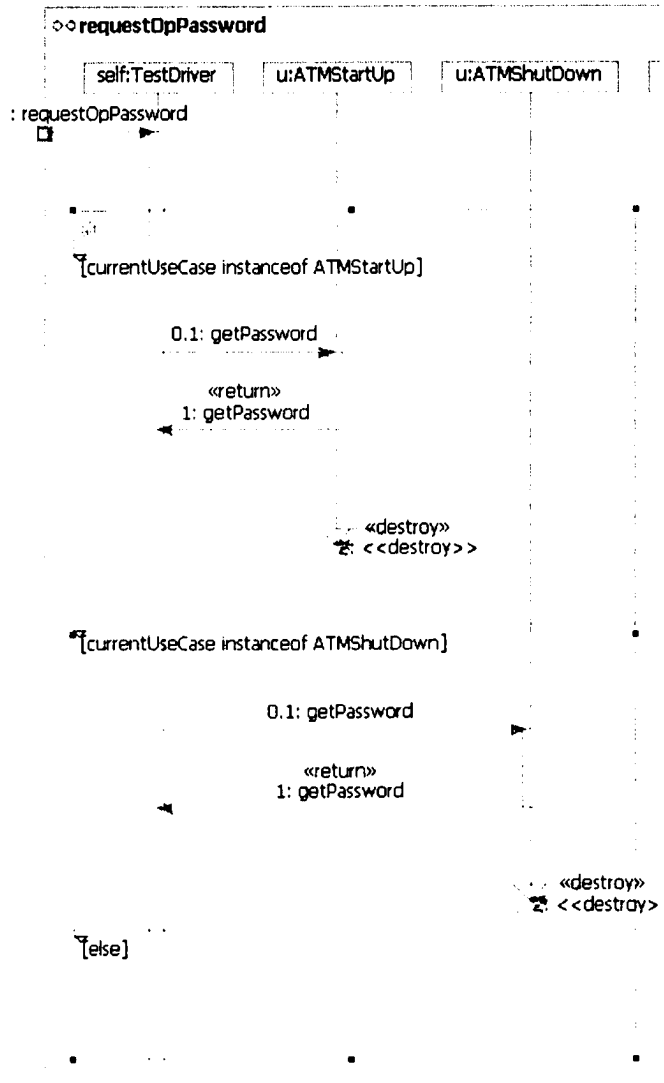

Figure 102 - requestOpPassword Partial Sequence Diagram

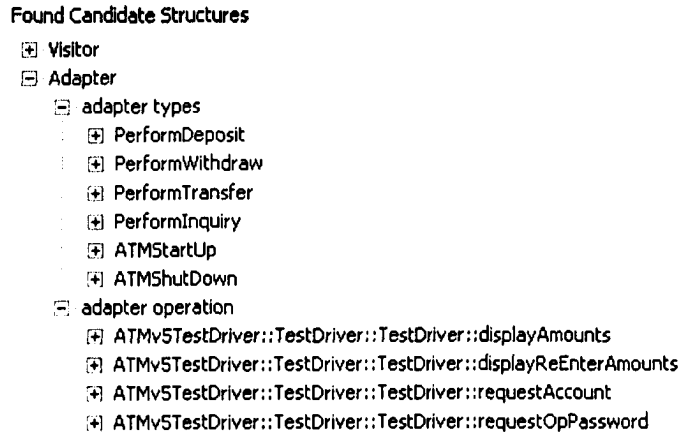

Figure 103 - DPA Result for *Adapter* Pattern

Although the conditional statements are similar to the candidate structures discussed in Section 7.2.3, the application of the *Adapter* pattern is not the ideal solution to this problem. A closer look at the conditional statements reveals that the methods being invoked have identical signatures and the types on which it is invoked all implement the `UseCase` interface. Table 7 provides a listing of the operations being invoked; using this information the original inheritance hierarchy can be modified to that shown in Figure 104. This new inheritance hierarchy would eliminate the need for the conditional statements and reduce code duplication.

Table 7 - Operations Invoked and Classes

Operation	Class
getAmount	PerformDeposit, PerformWithdrawl, PerformTransfer
getFromAccount	PerformDeposit, PerformWithdrawl, PerformTransfer, PerformInquiry
getPassword	ATMStartup, ATMShutdown

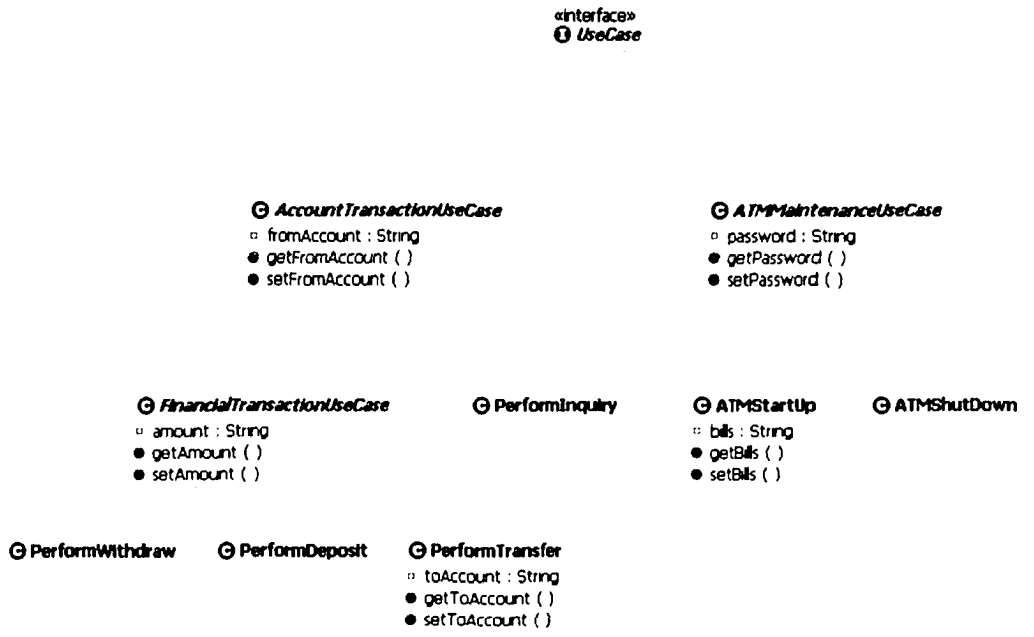

Figure 104 - Enhanced ATM UseCase Hierarchy

Although the suggestion offered by the DPATool to apply the *Adapter* pattern was not the correct action to be taken, it highlighted a flaw in the design of the system that should be addressed. This emphasizes that the DPATool is merely that: a tool that provides suggestions to the developers. Using the information regarding the suggested design pattern, provided by the DPATool, the developer is then able to determine what course action should be taken. This strengthens the argument for the semi-automated and heuristic approach taken in the DPATool.

Note that the decision trees developed can be incrementally improved. The decision tree can be modified to ensure that this false positive does not appear in future analysis. The specifications shown in Figure 60 can be enhanced to the decision tree shown in Figure 105 based on the findings of this case study.

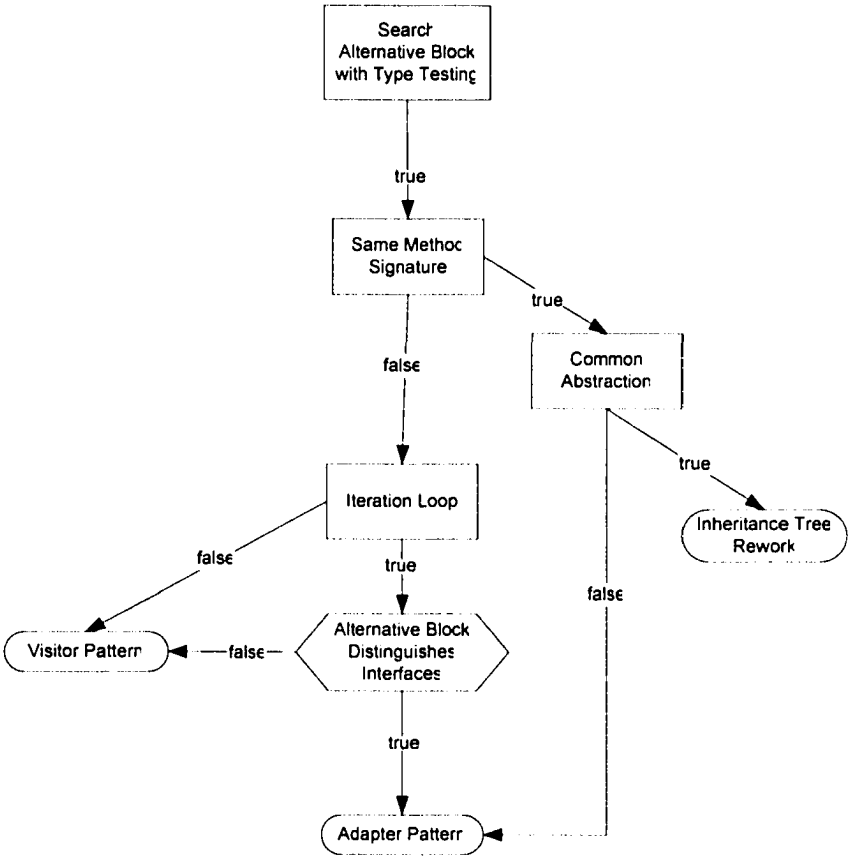

Figure 105 - *Visitor/Adapter* Decision Tree Enhanced

A textual description of the newly added nodes in this decision tree is given below and a formal specification in OCL is given in Appendix K

- *Same Method Signature:* Check to determine if the methods being invoked inside the alternative combined fragment have the same signature.
    - *in combFrag:CombinedFragment* - the alternative combined fragment being investigated.

- *Common Generalization:* Check to determine if the classes being used share a common super type or interface.
    - *in combFrag:CombinedFragment* - the alternative combined fragment being investigated.

- *Inheritance Tree Rework:* Suggestion to explore the rework of the inheritance tree or interface use to eliminate the need for the alternative combined fragment.

The solution when suggesting the *Inheritance Tree Rework*[9] pattern would be similar to the solution proposed earlier.

The example enhancement of the decision trees shown here emphasizes that the decision trees developed in this work will be incrementally refined. The decision trees provided here are not intended to be the final product, rather they are the beginning of a specification that will be developed and refined through empirical means.

10.4 Summary

The DPATool identified a clear potential use of the *State* pattern on the `Elevator` class in the Elevator Control System. In the case of the ATM Test Driver case study the DPATool provided constructive feedback for the application of both the *Visitor* and *Factory Method* patterns. The *Visitor* pattern could be applied to remove five if-then-else blocks in the `TestDriver` class. The *Factory Method* was suggested to replace the if-then-else block found in the `parseUseCases` method of the `TestCase` class. The DPATool also suggested the application of the *Adapter* pattern to a suspect series of if-then-else blocks found also in the `TestDriver` class. This set of if-then-else blocks did require some refactoring; however the application of the *Adapter* pattern was not the ideal solution. A refinement to the decision tree used conducted to eliminate this false positive; clearly demonstrating that the approach (decision tree and tool) can be easily updated to account for new empirical findings. Those case studies indicate that our approach shows potential for opening an area of research on better applying design patterns during design which has been overlooked.

---

[9] Technically this would not be considered a design pattern, but rather is it a design convention that should be applied.

11 CONCLUSION

This research has proposed an approach to help developers make better use of design patterns. The approach provides a means to detect UML structures, referred to as candidate structures, on which design patterns should be applied. These structures exists in modern software systems for a variety of reasons: for instance, the shear size and complexity of the systems makes it difficult to identify every possible area for improvement in a design, the vast number of design patterns available make it difficult for a designer to be knowledgeable in all the design patterns.

The approach we considered in this thesis consists of developing detection rules in the form of a decision tree that identifies these candidate structures. The decision tree provides an excellent structure for this form of analysis: The nodes represent the stepwise analysis that must be done against the UML model [41] to detect the candidate structures; The arcs between the nodes represent various navigation paths, where the path chosen is dependent on the result of the analysis performed; Finally, the leave nodes represent suggestions for the application of a particular pattern to the identified structure(s). Each time an analysis of the UML model under study is necessary, it is formalized in OCL [40] so as to be unambiguous.

Decision trees were developed for various Gang of Four design patterns: *State*, *Visitor*, *Adapter*, *Factory Method*, *Abstract Factory*, and *Decorator* [26]. This allowed us to determine the feasibility of our approach with the information available through UML 2 documentation. It was evident at the early stages of the thesis that some information would not be available through the UML documentation; therefore a semi-automated approach was implemented where the user can be queried during the decision process for the required information. The identification of pieces of information that could only be provided by the user was facilitated by our formalization of detection rules in OCL as it was then clear that the UML 2 metamodel elements as used in the OCL expressions were sometimes not sufficient.

With the rules formalized in OCL, a prototype tool supporting our approach (referred to as DPATool) was developed using the Eclipse Platform and other IBM technology to analyze UML models. The DPATool makes use of the Eclipse Modeling Framework (EMF) to model the decision trees. The EMF technology was used as the foundation for implementing the Eclipse UML 2 project to instantiate the UML 2.0 models to be analyzed. IBM's OCL Engine provided a means to automatically evaluate OCL expressions against an EMF model object (in our case a UML model element). This is done by using an ANTLR generated parser to create an AST tree of the OCL expression and EMF's reflection capabilities to perform the necessary operations. The OCL Engine was also enhanced to allow the decision tree designer to extend the UML2 metamodel where required. This was done by providing extensions to the OCL Engine and using reflection to invoke those extensions where required. The DPATool allows for quick development of decision trees using OCL expressions, Java, and user queries. The DPATool was also designed to be extensible by modifying existing or adding new types of analysis performed at the nodes of the decision trees. The DPATool was also designed to be integrated into a variety of case tools.

The DPATool was used in an empirical case study to test the approach we developed against two existing software systems: the ATM Test Driver and the Elevator Control System. For both these systems UML models were available and analyzed using the implemented decision trees. The DPATool identified a clear potential use of the *State* pattern in the Elevator Control System. In the case of the ATM Test Driver case study the DPATool provided constructive feedback for the application of both the *Visitor* and *Factory Method* patterns. The DPATool also suggested the application of the *Adapter* pattern to a suspect series of if-then-else blocks in the ATM Test Driver. This set of if-then-else blocks did require some refactoring; however the application of the *Adapter* pattern was not the ideal solution. A refinement to the decision tree used conducted to eliminate this false positive; clearly demonstrating that the approach (decision tree and tool) can be easily updated to account for new empirical findings. Those case studies indicate that our approach shows potential for opening an area of research on better

applying design patterns during design which has been overlooked. As well this research coupled with that of O Cinnéide [38] in some cases (the transformations supported by O Cinnéide's research) can provide an end to end solution for the application of design patterns.

This research was however limited in terms of empirical evaluation. First, the UML 2.0 specifications have only been finalized recently and as such the tools and thus the systems (case studies) currently available that use this specification are limited. More evaluation on existing systems will be needed to ascertain the contribution of this research. Further evaluation would indicate more accurately the completeness and accuracy of the decision trees developed. By gathering further data on the accuracy and completeness of the decision trees further refinements can be made as was done for the *Adapter* pattern tree in the case study.

With more evaluations the scalability of this approach and the algorithms for each decision tree can also be investigated against larger models. The specification section discusses the scalability for each of the decision trees developed; however these algorithms were not enhanced for performance. If the DPATool is found to have a low performance, a refinement of the decision trees can be done to improve their efficiency. This was not done presently as it is not certain that such performance refinements will be necessary.

To further evaluate the approach, the DPATool itself can be made more user-friendly. This would allow for more decision trees to be generated for other existing patterns. The improvements required would include better debugging for the analysis in the decision trees, a more powerful graphical editor for the decision trees, and more flexibility when querying the user. To enhance the decision process, a future feature can track the accuracy of the suggestions made to the user. The ratio of constructive suggestions versus false positives can then be transformed into probabilities associated to the leaf nodes of the decision trees: this would give the user a probability measure that a design pattern should be applied on the identified candidate structure(s). This will guide the

designers of the decision trees to areas needing further refinements and help the users of the decision trees assess the accuracy of the suggestion from the DPATool.

12 REFERENCES

[1] Meta Programming, http://c2.com/cgi/wiki?MetaProgramming, (Last accessed 2005 May 4)

[2] S. S. Alhir, *UML in a Nutshell*, O'Reilly & Associates, 1998.

[3] S. Ambler, "Java Programming Guidelines," Ambysoft Inc., technical report, http://www.ambysoft.com/javaCodingStandards.pdf, 2000.

[4] K. Beck, *eXtreme Programming eXplained: Embrace Change*, Addison-Wesley, 2000.

[5] Borland, Together, http://www.borland.com/together/, (Last accessed February 23 2005)

[6] J. Brant and D. Roberts, Refactoring Browser, http://st-www.cs.uiuc.edu/users/brant/Refactory/, (Last accessed January 14 2004)

[7] L. Briand, "SYSC 4800 - Software Engineering Course Notes," in *Extended State Pattern*. Carleton University, Department of Systems and Computer Engineering, 2005.

[8] L. Briand, Y. Labiche, K. Buist and G. Soccar, "Automating Impact Analysis and Regression Test Selection Based on UML Designs," Carleton University TR SCE-02-04, http://www.sce.carleton.ca/squall/pubs/tech_report/TR_SCE-02-04.pdf, 2003.

[9] B. Bruegge and A. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns, and Java*, Pearson Prentice Hall, 2nd Edition, 2004.

[10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *Pattern-Oriented Software Architecture*, Software Design Patterns, vol. 1, John Wiley & Sons, 1996.

[11] J. Coplien, "Software Design Patterns: Common Questions and Answers," AT&T Bell Laboratories, ftp://st.cs.uiuc.edu/pub/patterns/papers/PatQandA.ps.

[12] Eclipse Foundation, Business Intelligence and Reporting Tools Project, http://www.eclipse.org/birt/, (Last accessed May 1 2005)

[13] Eclipse Foundation, Eclipse Modeling Framework, http://download.eclipse.org/tools/emf/scripts/home.php, (Last accessed April 25 2005)

[14] Eclipse Foundation, Eclipse Project, www.eclipse.org, (Last accessed May 2 2005)

[15] Eclipse Foundation, Eclipse Technology Project, http://www.eclipse.org/technology/index.html, (Last accessed May 1 2005)

[16] Eclipse Foundation, Eclipse Test and Performance Tools Platform Project, http://www.eclipse.org/tptp/index.html, (Last accessed May 1 2005)

[17] Eclipse Foundation, Eclipse Tools Project, http://www.eclipse.org/tools/index.html, (Last accessed May 5 2005)

[18] Eclipse Foundation, Eclipse Web Tools Platform Project, http://www.eclipse.org/webtools/index.html, (Last accessed May 1 2005)

[19] Eclipse Foundation, Java Development Tools Subproject, http://www.eclipse.org/jdt/index.html, (Last accessed May 1 2005)

[20] Eclipse Foundation, Plug-in Development Environment Subproject, http://www.eclipse.org/pde/index.html, (Last accessed May 1 2005)

[21] Eclipse Foundation, UML2: EMF-Based UML 2.0 Metamodel Implementation, http://www.eclipse.org/uml2/, (Last accessed May 1 2005)

[22] A. Eden, "A visual formalism for object-oriented architecture," *Proc. Integrated Design and Process Technology*, 2002.

[23] A. Eden, A. Yehudai and J. Gil, "Precise Specification and Automatic Application of Design Patterns," *Proc. Automated Software Engineering*, 1997.

[24] M. Fowler and K. Scott, *UML Distilled*, Addison-Wesley, 2nd Edition, 2000.

[25] Free Software Foundation, CVS - Concurrent Versions System, http://www.gnu.org/software/cvs/, (Last accessed November 25 2004)

[26] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Professional Computing Series, Addison-Wesley, 1995.

[27] E. Gamma, L. Nackman and J. Wiegand, *Eclipse Modeling Framework*, the eclipse series, Addison-Wesley, 2004.

[28] Y. G. Guéhéneuc, *Un cadre pour la tracabilite des motifs de conception*, Thesis, Universite de Nantes, Ecole doctoral Sciences et Technologies de l'Information et des Materiaux, 2003

[29] Y. G. Guéhéneuc and H. Albin-Amiot, "Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-class Design Defects.," *Proc. TOOLS USA*, 2001.

[30] I. Hamid, I. Khriss and R. Keller, "Supporting Design by Pattern-based Transformations," Universite de Montreal, technical report, 1999.

[31] IBM, Model Transformation Framework, http://www.alphaworks.ibm.com/tech/mtf, (Last accessed May 3 2005)

[32] J. Kerievsky, *Refactoring to Patterns (DRAFT)*, Addison-Wesley, 2003.

[33] S. F. Khalid and J. Rowat, "Design of an Elevator Control System," Carleton University, 4th year project, 2004.

[34] P. Kuchana, *Software Architecture Design Patterns in Java*, Auerbach Publications, 2004.

[35] Microsoft, Microsoft Office Online: FrontPage 2003 Home Page, www.microsoft.com/frontpage/, (Last accessed May 27 2005)

[36] S. Microsystems, Collection (Java 2 Platform SE v1.4.2), http://java.sun.com/j2se/1.4.2/docs/api/java/util/Collection.html, (Last accessed 2005 May 1)

[37] T. Muraki and M. Saeki, "Metrics for Applying GOF Design Patterns in Refactoring Processes," *Proc. International Workshop Principles of Software Evolution*, Vienna Austria, ACM, pp. 27-36, 2001.

[38] M. O'Cinnéide, *Automated Application of Design Patterns: A Refactoring Approach*, PhD Thesis, University of Dublin, Department of Computer Science, 2000

[39] M. O'Cinnéide and P. Nixon, "Automated Software Evolution Towards Design Patterns," *Proc. International Workshop on Principles of Software Evolution*, Vienna Austria, ACM, pp. 162-165, 2001.

[40] OMG, "Reponse to the UML 2.0 OCL RFP," Object Management Group, 2003.

[41] OMG, "UML 2.0 Superstructure Specification - Final Adopted Specification," Object Management Group, 2003.

[42] Omondo, Omondo Studio, http://www.omondo.com/, (Last accessed April 15 2005)

[43] Rational-IBM, Rational Clearcase, http://www-306.ibm.com/software/awdtools/clearcase/, (Last accessed 2005 January 22)

[44] Rational-IBM, Rational Software Architect, http://www-306.ibm.com/software/awdtools/architect/swarchitect/, (Last accessed April 23 2005)

[45] D. Roberts, *Eliminating Analysis in Refactoring*, PhD Thesis, University of Illinois, Department of Computer Science, 1999

[46] Sun Microsystems, Java API Specifications, http://java.sun.com/j2se/1.4.2/docs/api/index.html, (Last accessed May 5 2005)

[47] Sun Microsystems, java.lang.reflect (Java 2 Platform SE v1.4.2), http://java.sun.com/j2se/1.3/docs/api/java/lang/reflect/package-summary.html, (Last accessed March 21 2005)

[48] Sun Microsystems, Pattern (Java 2 Platform SE v1.4.2), http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html, (Last accessed May 3 2005)

[49] L. Tokuda and D. Batory, "Automated Software Evolution via Design Pattern Transformations," *Proc. International Symposium on Applied Corporate Computing*, Monterrey, Mexico, 1995.

[50] L. Tokuda and D. Batory, "Evolving object-oriented designs with refactorings," *Proc. Automated Software Engineering*, 1999.

[51] W. Zimmer, B. Schulz, T. Genssler and B. Mohr, "On the computer aided introduction of design patterns into object-oriented systems," *Proc. TOOLS*, 1998.

Appendix A Complete Meta-Model

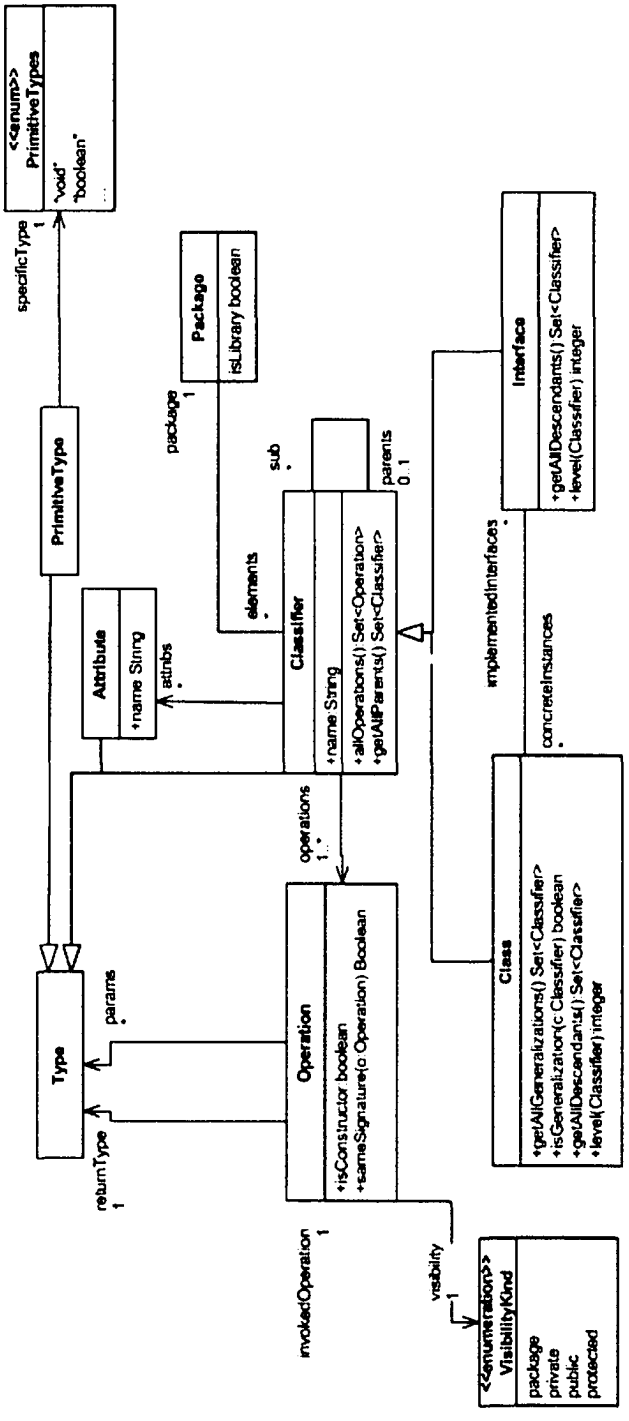

Figure 106 – (Part A – Static) Complete Meta-Model

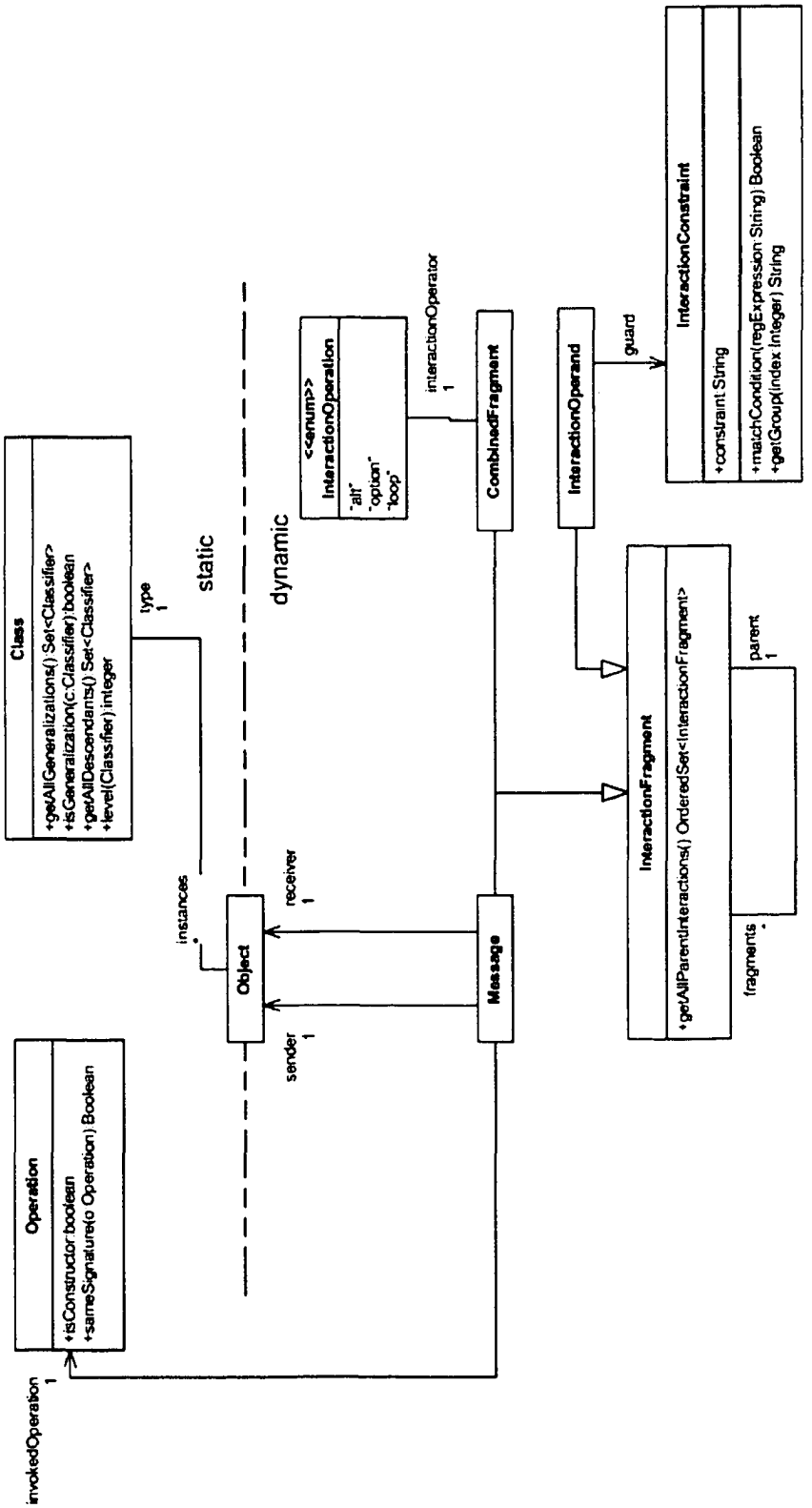

Figure 106 – (Part B – Dynamic) Complete Meta-Model

Appendix B Meta-Model Example

This is an example instantiation of the meta-model presented in Figure 106.

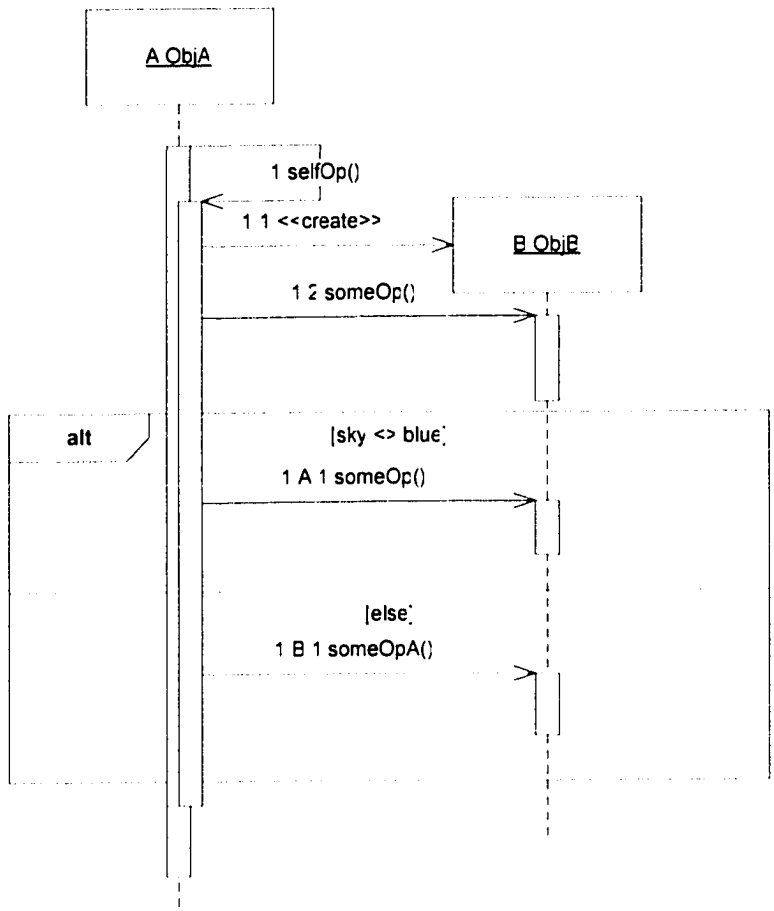

Figure 107 - Meta-Model Example Sequence Diagram

Figure 108 - Meta-Model Example Instantiation

Appendix C OCL Rules for Factory Method and Abstract Factory Patterns

```
-- Ensure that the combined fragment is an alternative.
node Alternative_Combined_Fragment(
  in combFrag:CombinedFragment
):Boolean
post:
  result = combFrag.interactionOperator = InteractionOperator::alt

-- Node ensures that each of the interation operand within
-- the combined fragment contain constructors.
-- combFrag - the combined fragment to be tested.
node Alternative_Fragments_Contain_Constructors(
  in combFrag:CombinedFragment
):Boolean
post:
  -- Check for constructor in each of the InteractionFragment operands within
  -- the combined fragment.
  result = combFrag.fragments->forAll(op:InteractionOperand|
      op.fragments->exists(i:InteractionFragment|
        i.oclIsTypeOf(Message)
        and
        i.operation.isConstructor
      )
    )

-- Ensure that for all the classes that are instantiated
-- there is a common generalization that they all share
-- which is not trivial
-- combFrag - the combined fragment being tested.
-- trivialGeneralizations - a list of classifiers that would be
--   a trivial generalization.
-- generalizations - the set of generalizations that are shared
--   the classes created between the InteractionFragment operands.
node Common_Generalization(
  in combFrag:CombinedFragment
  in trivialGeneralizations:Set<Classifier>
  out generalizations:Set<Classifier>
):Boolean
post:
  generalizations =
  Classifier.allInstances->excluding(trivialGeneralizations)->select(c:Classifier|
    combFrag.fragments->forAll(op:InteractionOperand|
      op.fragments->exists(i:InteractionFragment|
        i.oclIsTypeOf(Message)
        and
        i.operation.isConstructor
        and
        c.isGeneralization(i.operation.type)
        )
    )
  )
  and
  if generalizations->size() > 0 then
    result = true
  else
    result = false
  endif
```

```
-- Ensure that the method containing the combined fragment
-- is not returning one of the common generalizations
-- combFrag - the combined fragment being tested.
-- generalizations - the set of generalizations that are shared
--   the classes created between the InteractionFragment operands.
node Returns_Common_Generalization(
  in combFrag:CombinedFragment
  in generalizations:Set<Classifier>
):Boolean
post:
  result = generalizations->includes(combFrag.getAllParentInteractions()-
>select(inter:InteractionFragment| inter.oclIsTypeOf(Message))->first().returnValue)

-- Used to determine if there are multiple families being constructed in the
-- combined fragment.
-- combFrag - the combined fragment being tested.
node Multiple_Families(
  in combFrag:CombinedFragment
):Boolean
post:
  let forbidClassifiers:Set(Classifier) =
    combFrag.fragments.fragments->select(i:InteractionFragment|
      i.oclIsTypeOf(Message)
      and
      i.operation.isConstructor
    ).operation.classifier->asSet.forAll(c1, c2:Classifier|
      result->includesAll(c1.getAllGeneralizations()->
        intersection(c2.getAllGeneralizations())
      )
    )
  in
  if (
    combFrag.fragments->first->fragments->select(i:InteractionFragment|
      i.oclIsTypeOf(Message)
      and
      i.operation.isConstructor
    )->forAll(m1:Message|
      Classifier.allInstances->excluding(forbidClassifiers)
      ->exists(c:Classifier|
        c.isGeneralizationOf(m1.receiver.type)
        and
        Sequence{2..combFrag.fragments->size}->forAll(n:Integer|
          combFrag.fragments->at(n).fragments->exists(i2:InteractionFragment|
            i2.oclIsTypeOf(Message)
            and
            i2.operation.isConstructor
            and
            c.isGeneralizationOf(i2.operation.classifier)
          )
        )
      )
    )
  ) then
        result = true
  else
        result = false
```

```
-- Checks if all the classes created within this combined fragment
-- are from the same package.
-- combFrag - the combined fragment being tested.
-- trivialGeneralizations - a list of classifiers that would be
--   a trivial abstraction.
node Same_Package(
  in combFrag:CombinedFragment
):Boolean
post:
  -- Try to see if the objects that are created are all found in the same
  -- package.  This will be the first indication that the classes created in the
  -- combined fragments are from a library package.
  let packages = combFrag.fragments.fragments->select(i:InteractionFragment |
      i.oclIsTypeOf(Message)
      and
      i.operation.isConstructor
    ).receiver.type.package->asSet()
  in
  if packages->size() = 1 then
    result = true
  else
    result = false
  endif

-- Determines if there are similar combined fragments found in the
-- system.  The test for similarity is just to ensure that there
-- are the same number of InteractionFragment operands and that the
-- classes created are the same.
-- combFrag - the combined fragment being tested
-- similarCombFrags - a Set of the combined fragments that are deemed
--   to be similar to combFrag.
node Similar_Alternative_Block(
  in combFrag:CombinedFragment
  out similarCombFrags:Set<CombinedFragments>
):Boolean
post:
  similarCombFrags = CombinedFragments.allInstances->select(simCombFrag|
    simCombFrag <> combFrag
    and
    combFrag.fragments->forAll(io1:InteractionOperand|
      io1.fragments->exists(i1:InteractionFragment|
        i1.oclIsTypeOf(Message)
        and
        i1.operation.isContructor
        and
        simCombFrag.fragments->exists(io2:InteractionOperand|
          io2.fragments->exists(i2:InteractionFragment|
            i2.oclIsTypeOf(Message)
            and
            i2.operation.isContructor()
            and
            i2.receiver.type = i1.receiver.type
          )
        )
      )
    )
  )
  and
  if similarCombFrags->size() > 0 then
    result = true
  else
    result = false
  endif
```

```
-- Once the similar combined fragments are discovered then
-- determine if it is there are multiple clients.
-- combFrag - the combined fragment being tested.
-- similarCombFrags - a Set of the combined fragments that are deemed
--   to be similar to combFrag.
--   Provided by the Find_Similar_Alternative_Block node.
node Same_Client(
  in combFrag:CombinedFragment
  in similarCombFrags:Set<CombinedFragment>
):Boolean
post:
  if similarCombFrags->exists(simCombFrag:CombinedFragment|
      simCombFrag.fragments.fragments->exists(i:InteractionFragment|
        i.oclIsTypeOf(Message)
        and
        i.operation.isConstructor
        and
        combFrag.fragments.fragments->exists(i2:InteractionFragment|
          i2.oclIsTypeOf(Message)
          and
          i2.operation.isConstructor
          and
          i.sender.type = i2.sender.type
        )
      )
    )
  then
    result = true
  else
    result = false
  endif
```

Appendix D OCL Rules for Observer Pattern

```
-- Locate method call with a subsequent method call back to the original
-- sender
-- observerInteraction - the message being investigated
node Observer_Method_Call(
  in observerInteraction:Message
):Boolean
post:
  if
    -- the method call to observer does not change the observer's state
    observerInteraction.invokedOperation.isQuery()
    and
    -- void notify method - if it is a simple notification of a state change
    -- why would there be a value returned?
    observerInteraction.operation.returnType.oclIsTypeOf(PrimitiveType)
    and
    observerInteraction.operation.returnType.specificType = PrimitiveType::void
    and
    -- there immediately follows a query back to the sender
    observerInteraction.fragments->exists(i:InteractionFragment |
      i.oclIsTypeOf(Message)
      and
      i.receiver.type = observerInteraction.sender.type
      and
      i.invokedOperation.isQuery()
    )
  then
    result = true
  else
    result = false
  endif

-- Determine if the observerInteraction appears inside a loop structure.
-- observerInteraction - the message being investigated
node Locate_Loop(
  in observerInteraction:Message
):Boolean
post:
  result = observerInteraction.getAllParentInteractions()->exist(i:InteractionFragment|
i.oclIsTypeOf(CombinedFragment) and i.interactionOperator = InteractionOperator::loop)

-- Determine if the observerInteraction is located within a self-invocation.
-- observerInteraction - the message being investigated
-- selfCall - the self invocation if found
node Locate_Self_Call(
  in observerInteraction:Message
  out selfCall:Message
):Boolean
begin:
  let parentMethodInv = observerInteraction.getAllParentInteractions()-
>select(i:InteractionFragment| i.oclIsTypeOf(Message))->last()
  in
  if parentMethodInv.sender = parentMethodInv.receiver then
      result = true
      selfcall = parentMethodInv
    else
      result = false
  endif
```

```
-- Determine if the type of object receiving the observerInteraction message
-- is being abstracted by an interface.
-- observerInteraction - the message being investigated
-- observerInterface - the interface that is used for the observer type
node PseudoObserver_Interface(
  in observerInteraction:Message
  out observerInterface:Interface
): Boolean
post:
  if observerInteraction.receiver.type.oclIsTypeOf(Interface) then
    observerInterface =  observerInteraction.receiver.type
    result = true
  else
    -- Test to see if the concrete type has an interface that defined
    -- the function called initially
    -- Store this value for use in the PseudoSubject_Implementation_Class
    -- procedure
    observerInterface = observerInteraction.receiver.type.allGeneralizations()-
>select(generalization|
      generalization.oclIsTypeOf(Interface)
      and
      generalization.operations->includes(observerInteraction.invokedOperation)
      )->asSequence->first
    and
    if observerInteraction <> null then
      result = true
    else
      result = false
    endif
  endif

-- Determine if the sender of the observerInteraction makes use of an interface
-- or abstract class to define the necessary procedures for the Observer
-- pattern.
-- observerInteraction - the message being investigated
-- observerInterface - the interface that is used for the observer type
--    Provided by the PseudoObserver_Interface node
-- selfCall - the self invocation if found
--    Provided by the Locate_Self_Call node
node PseudoSubject_Implementation_Class(
  in observerInteraction:Message
  in selfCall:Message
  in observerInterface:Classifier
):Boolean
begin:
  -- Try to find an generalization that defined the self-call operation and
  -- an operation that take the observer interface type as a parameter.
  -- This is to ensure that the observer's have a means to subscribe to the
  -- subject.
  if observerInteraction.sender.type.allGeneralizations()->exists(generalization|
      generalization.operations->includes(selfCall.invokedOperation)
      and
      generalization.operations.params->includes(observerInterface)
      )
  then
    result = true
  else
    result = false
  endif
```

Appendix E OCL Rules for Decorator Pattern

```
-- Test the hierarchy size to see if it meets the requirements for further
-- investigation.
-- hierarchySize - the size of the hierarchies to be considered
--
node Search_Large_Hierarchy(
   in hierarchySize:Integer
   in classifier:Classifier
   in trivialGeneralizations:Set<Classifier>
   out rootClassifier:Classifier
):Boolean
post:
  if
    classifier.getAllDescendants()->exclude(d | d.oclIsTypeOf(Interface))->size() >
hierarchySize
  then
    rootClassifier = classifier
    result = true
  else
    result = false
  endif

-- Attempts to determine if the interfaces for the classes change through
-- the hierarchy.
-- interfaceRatio - the ratio of inherited public methods versus the newly
--    defined public methods that is considered to be a close compliance
-- rootClassifier - the root of the inheritance tree being examined
node Similar_Interfaces(
  in interfaceRatio:Double
  in rootClassifier:Classifier
):Boolean
post:
  if rootClassifier.sub->size() = 0 then
    result = true
  else
    if rootClassifier.oclIsTypeOf(Interface) then
      result = rootClassifier.sub->forAll (child |
          Similar_Interfaces(interfaceRatio, child))
  else
    result = rootClassifier.sub->forAll(child |
      (rootClassifier.allOperations()->intersection(child.allOperations())
        ->size() / child.allOperations()->size()) < interfaceRatio
      and
      Similar_Interfaces(interfaceRatio, child))
  endif

-- Checks if the hierarchy can inform whether the functionality is
-- independent.
-- rootClassifier - the root of the inheritance tree being examined
node Scan_Hierarchy_Independent_Functionality(
  in rootClassifier:Classifier
):Boolean
post:
  result = rootClassifier.allDescendants().level(rootClassifier)->asSet()->
    forAll(i:Integer|rootClassifier.allDescendants()->select(d:Classifier|
      d.level = i)->forAll(d1, d2:Classifier|d1.sub->size() = d2.sub->size()))
```

Appendix F OCL Rules for Visitor and Adapter Pattern

```
-- Check if the alternative combined fragment makes use of type testing in
-- the guard conditions.
-- combFrag - the alternative combined fragment being investigated
node Check_Alternative_Block_Type_Testing(
  in combFrag:CombinedFragment
):Boolean
post:
  result = combFrag.fragments->forAll(io:InteractionOperands|
    io.guard.matchCondition('.*\W+instanceof\W+.*'))

-- Check if there is an iteration loop surrounding this
-- alternative combined fragment.
-- combFrag - the alternative combined fragment
-- keywords - the set of words that may indicate that is
--  used as a iteration loop.  These keywords are already
--  in the regular expression pattern.
node Iteration_Loop(
  in combFrag:CombinedFragment
  in keywords:Set<String>
):Boolean
post:
  -- no need to go above the direct parent of this structure
  -- if it is used as iteration loop there should not be
  -- another alternative combined fragment surrounding this one
  result = combFrag.parent.oclIsTypeOf(CombinedFragment)
    and
    combFrag.parent.interactionOperator = InteractionOperator::loop
    and
    keywords->exists(keyword|
      combFrag.parent.fragments->exists(frag:InteractionOperand|
frag.guard.matchCondition(keyword))
```

Appendix G OCL Rules for State Pattern

```
-- Test to see if combined fragment's first interaction operand uses
-- a class attribute in the guard condition
-- combFrag - the combined fragment being investigated
-- alt2bChecked - the combined fragment to be tested to ensure that
--    the attribute is used guard conditions for each of the interaction operands
-- foundAttribs - the attibutes found in the guard condition
node Combined_Fragment_Using_Class_Attribute{
  in combFrag:CombinedFragment
  out foundAttribs:Set<Attribute>
  out alt2bChecked:CombinedFragment
}:Boolean
post:
  if (combFrag.fragments->size() > 1)
    and
    (combFrag.owner.type.attribs->select(a| a.oclIsTypeOf(PrimitiveType)).name->
      exists(n:String| combFrag.fragments-
>first().matchCondition('.*\W+'.concat(n).concat('\W+.*'))))
  then
    foundAttribs = combFrag.owner.type.attribs->select(a|
      a.oclIsTypeOf(PrimitiveType)).name->exists(n:String|
        combFrag.fragments->
        first().matchCondition('.*\W+'.concat(n).concat('\W+.*')))
    result = true
    alt2bChecked = combFrag
  else
    result = false
  endif


-- Test to determine if an attribute or set of attributes are
-- used in the guard condition for each interaction operand
-- for a given combined fragment.
-- foundAttribs - the set of attributes to check the guard
--   conditions against.
-- alt2bChecked - the combined fragment being tested
node Check_Guard_Condition_Alternative_Fragments(
  in alt2bChecked:CombinedFragment
  in/out foundAttribs:Set<Attribute>
):Boolean
post:
  result = alt2bChecked.fragments->forAll(io:InterationOperand|
    foundAttribs->exists(attrib:Attribute|
      io.matchCondition('.*\W+'.concat(attrib.name).concat('\W+.*'))))
  foundAttribs = foundAttribs->select(attrib:Attribute|
    alt2bChecked.fragments->forAll(io:InteractionOperand|
      io.matchCondition('.*\W+'.concat(attrib.name).concat('\W+.*'))

-- Test to find other combined fragments using the same set of
-- attributes to determine the classe's behaviour.
-- combFrag - the original combined fragment under test
-- foundAttribs - the set of attributes used in the original
--    combined fragment.
node Check_Alternative_Blocks_Same_Attributes(
  in combFrag:CombinedFragment
  in foundAttribs:Set<Attribute>
):Boolean
post:
  result = CombinedFragment.allInstances->exists(cf|
    combFrag <> cf
    and
    combFrag.owner.type = cf.owner.type
    and
```

```
  cf.fragments->forAll(io:InteractionOperand|
    foundAttribs->exists(attrib|
    io.matchCondition('.*\W+'.concat(attrib.name).concat('\W+.*')))
)
```

Appendix H ATM Class Diagrams

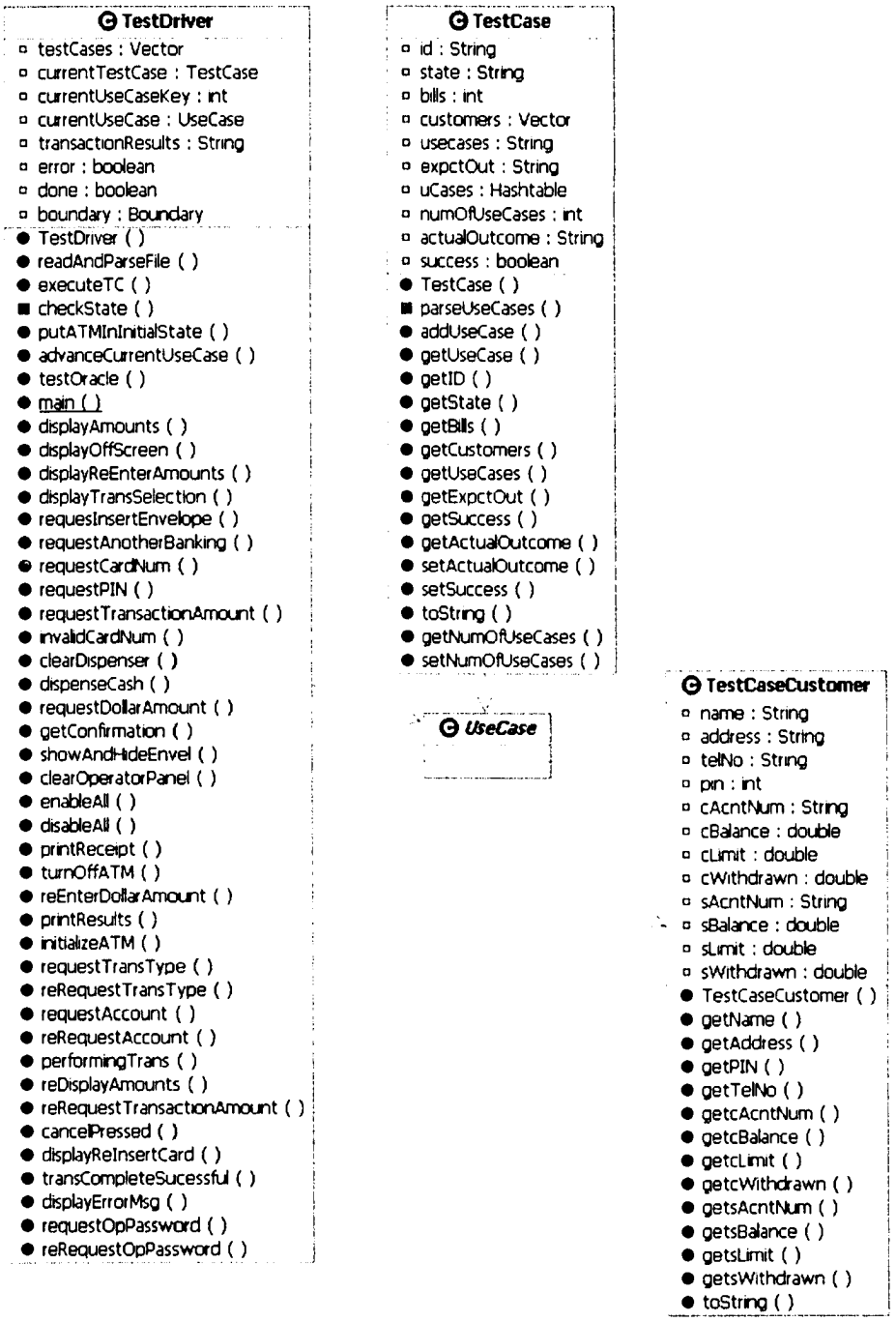

Figure 109 - ATM Overview Class Diagram

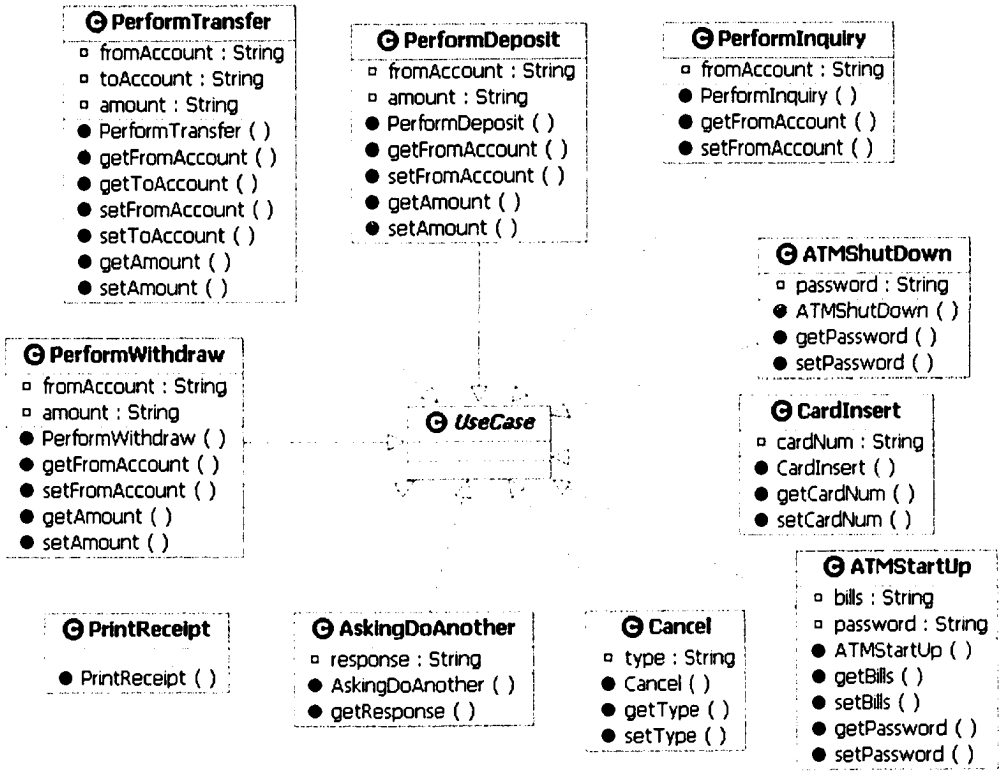

Figure 110 - ATM Use Case Hiearchy

Appendix I Elevator Control System Class Diagram

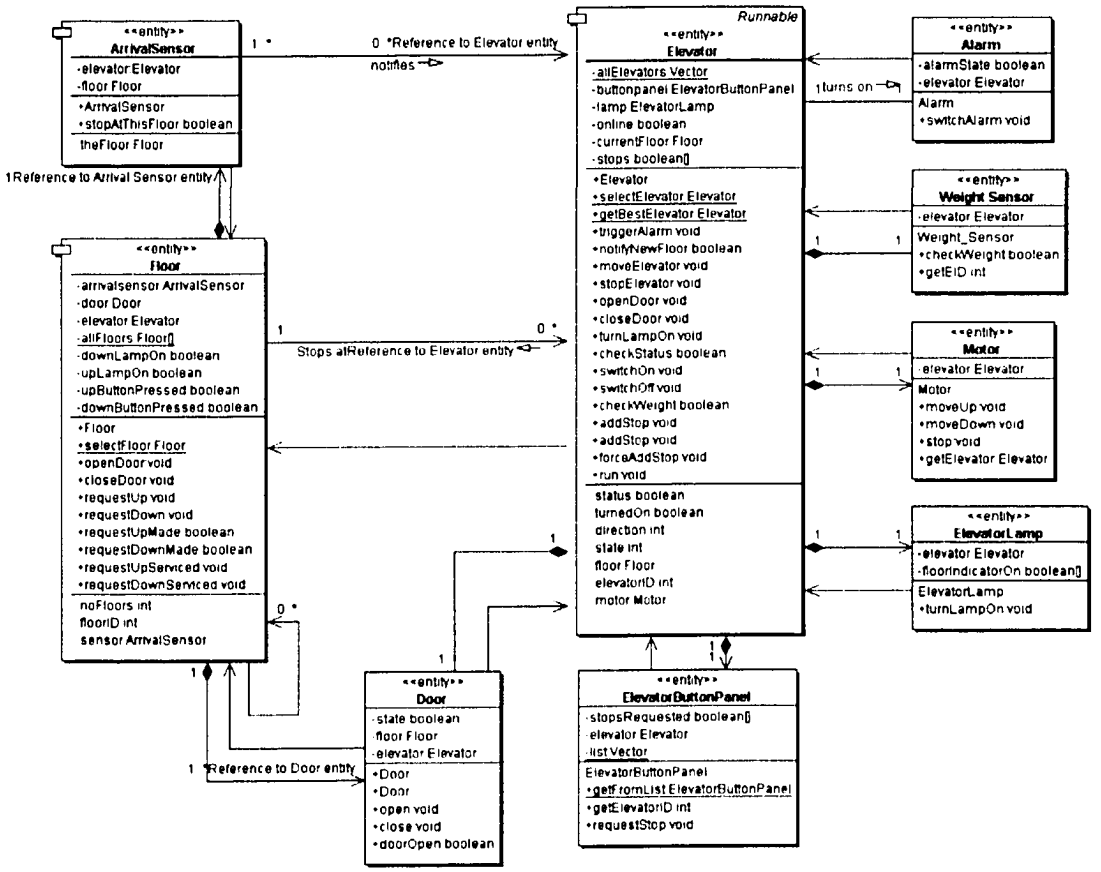

Figure 111 - ECS Entity Diagram

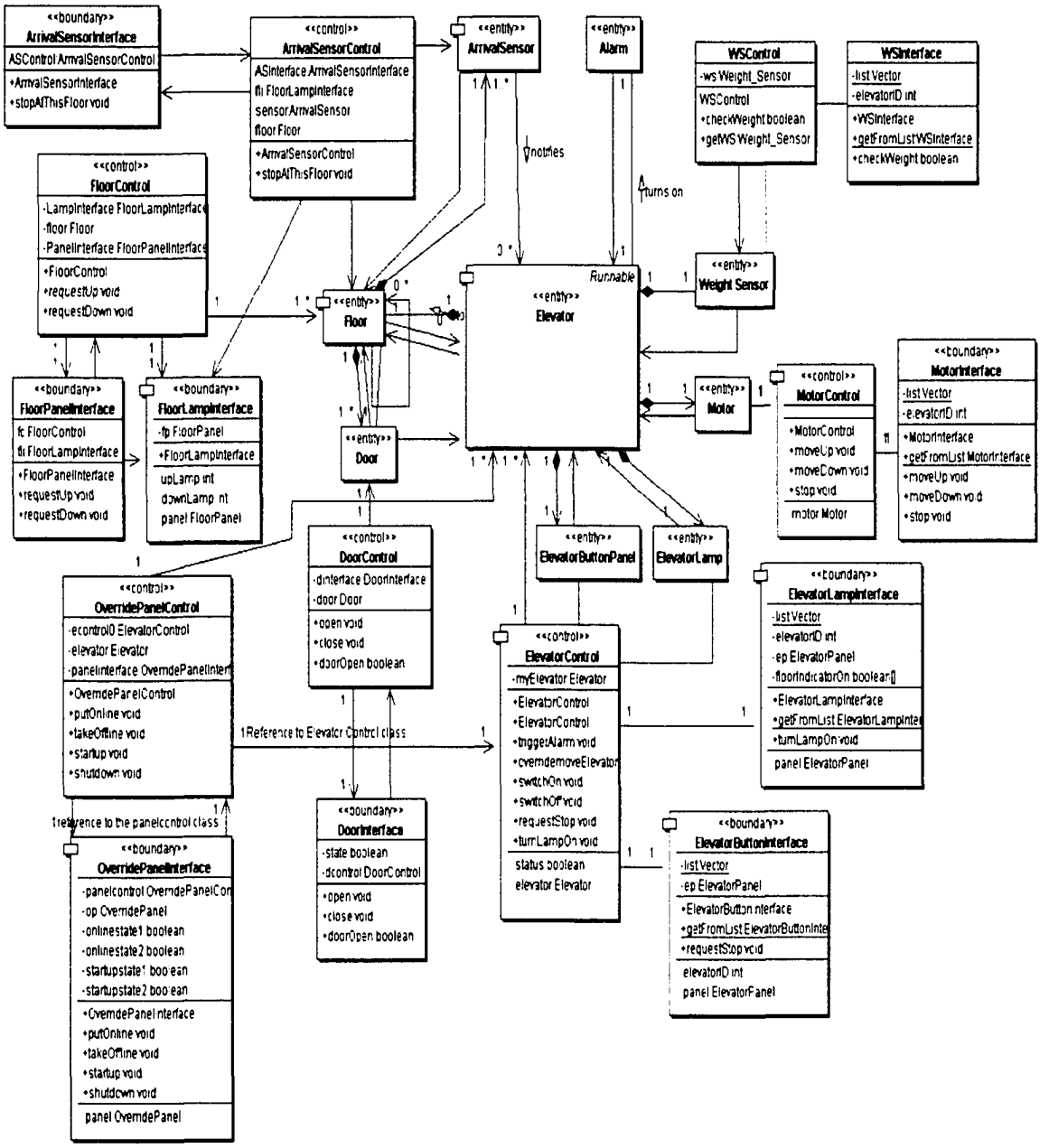

Figure 112 - ECS Complete Class Diagram

Appendix J moveElevator Source Code

```
/**
 * If there is a floor with a request in the current direction, move one floor
 * in the current direction. Else reverse direction and check again.
 */
public void moveElevator() {

  //Three-phase movement cycle. Prepare/move/check. Does not loop here,
  // since it returns control to run()
  //which calls this again if it hasn't set itself to Idle.

  //  PREPARE:
  if (this.state == PREPARE) {
    Tester.elevatorDisplay(elevatorID, "\n Door closed.  ");
    closeDoor();
    checkWeight();
    Tester.elevatorDisplay(elevatorID, " Weight checked.");
    this.state = MOVING;
    //System.out.println("\n Elevator "+(this.getElevatorID()+1)+" is
    // in state MOVING");

    if (direction == 1) {
      motor.moveUp();
    } else
      motor.moveDown();
  }
  while (this.state == MOVING) {
    //Do nothing - we're waiting to be interrupted by ArrivalSensor
    // telling us we've reached a new floor.
    try {
      Thread.sleep(500);
    } catch (Exception e) {
    }
  }
  while (this.state == FINDNEXT) {
    boolean directionReversed = false; //to make sure our loop isn't
    // endless
    int stopToCheck = currentFloor.getFloorID() + direction;
    while (Floor.selectFloor(stopToCheck) != null && state == FINDNEXT) {
      if (stops[stopToCheck] == true) {
        this.state = PREPARE;
        Tester.elevatorDisplay(elevatorID, "\n Next Stop = floor "
            + (stopToCheck + 1));
        //System.out.println("\n Elevator
        // "+(this.getElevatorID()+1)+" is in state PREPARE - line
        // 222");
      } else
        stopToCheck += direction;
    }
    if (Floor.selectFloor(stopToCheck) == null && !directionReversed
        && state == FINDNEXT) //If we ran out of floors before
    // finding a stop
    {
      direction = -direction; //reverse direction
      directionReversed = true; //Note that you've already checked.
    } else if (Floor.selectFloor(stopToCheck) == null && directionReversed
        && state == FINDNEXT) //If we ran out
    // of floors THE
    // OTHER WAY,
    // too.
    {
      this.state = Elevator.IDLE;
      Tester.elevatorDisplay(elevatorID, "\n All stops handled.  Idling.");
      //System.out.println("\n Elevator "+(this.getElevatorID()+1)+"
```

```
        // is in state IDLE - line 226");
        //Debugging flush of all stops:
        /*
         * for (int i=0;i <4 ;i++ ) { addStop(i,false); }
         */

      }//Idle again until called upon for another stop.
    }

  }
```

Appendix K New OCL Rules for Refined Visitor and Adapter Pattern

```
-- Check if the methods invoked on the object being tested have
-- the same signature.
-- combFrag - the alternative combined fragement under test
-- types - the types involved in the type check
node Same_Signature(
  in combFrag:CombinedFragment
  out types:Set<Classifier>
):Boolean
post:
-- gets the types of the classifiers being tested
let typesString = combFrag.fragments.guard->collect(guard:InteractionContraint|
guard.matchCondition("\w* instanceof (\w*)").getGroup(1)), types =
Classifier.allInstances->select(c:Classifier| typesString->includes(c.name))
in
  result = combFrag.fragments->forAll(io:InteractionOperand| combFrag.fragments-
>forAll(io1:InteractionOperand| io1 == io or io1.fragments-
>select(f1:InteractionFragment| f1.oclIsTypeOf(Message))->select(m1:Message| types-
>includes(m1.receiver.type)).invokedOperation->forAll(o1:Operation| io.fragments-
>select(f:InteractionFragment| f.oclIsTypeOf(Message))->select(m:Message| types-
>includes(m.receiver.type)).invokedOperation.sameSignature(o1))))

-- Check if the classes being tested have a common generalization.
-- types - the types involved in the type check
-- trivialGeneralizations - a list of classifiers that would be
--   a trivial generalization.
-- generalizations - the set of generalizations that are shared
--   the classes created between the InteractionFragment operands.
node Common_Generalization(
  in types:Set<Classifier>
  in trivialGeneralizations:Set<Classifier>
  out generalizations:Set<Classifier>
):Boolean
post:
  generalizations =
  Classifier.allInstances->excluding(trivialGeneralizations)->select(c:Classifier|
        types->forAll(type:Classifier|
          c.isGeneralization(type)
        )
    )
  )
  and
  if generalizations->size() > 0 then
    result = true
  else
    result = false
  endif
```