An Automated Approach to Object Oriented Design Pattern Detection and Extraction.

by

Jochen Stier
B.Sc., University of Victoria, 1994

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. H. Müller, Supervisor (Dept. of Computer Science)

[redacted]

Dr. D. Hoffman, Departmental Member (Dept. of Computer Science)

[redacted]

Dr. F. Diacu, Outside Member (Dept. of Mathematics)

[redacted]

Dr. Jochen Moehr, External Examiner (Dept. of Health Information Science)

Supervisor: Dr. H. Müller

ABSTRACT

Maintaining and re-engineering large software systems has proven to be extremely labor intensive, which is in part caused by a lack of knowledge about the system. Information gathering about design, structure and functionality often occupies a large portion of the maintenance efforts.

Object oriented design patterns describe common solutions to recurring problems in object oriented software systems. They are derived from the experience and expertise of partitioning software designers and developers. Literature about a pattern provides a vast pool of information ranging from pitfalls and advantages to detailed implementations. Each pattern encapsulates much more information about a system than the individual software abstractions composing it. Detection and presentation of the design patterns aids and enhances the information gathering process of the re-engineering and maintenance task by providing documentation about system architecture, design and behavior. A pattern furthermore provides a means of system decomposition as well as common terminology for system components.

This thesis presents a strategy to identify, extract and visualize object oriented design patterns that are present within the source code of an existing software system. The approach combines techniques of reverse engineering, querying and visualization to provide a tool for automated design pattern detection and extraction.

Examiners:

[redacted]

Dr. H. A. Müller, Supervisor (Dept. of Computer Science)

[redacted]

Dr. D. Hoffman, Departmental Member (Dept. of Computer Science)

[redacted]

Dr. F. Diacu, Outside Member (Dept. of Mathematics)

[redacted]

Dr. Jochen Moehr, External Examiner (Dept. of Health Information Science)

Table of Contents

List of Tables

List of Tables

List of Figures

Acknowledgments

I specially would like thank my supervisor, Dr. H. Müller, for supporting me as a graduate student and giving me free reign on how to go about my studies.

I also like to thank Dr. Dan Hoffman and Dr. Nigel Horspool. Both always had an open door whenever I needed some advice on C++ or Prolog. Dan Hoffman's 'pattern discussion group' was invaluable in strengthening my understanding of object oriented design patterns.

I would also like to thank Jim Uhl, who often provided me with suggestions and offered support when I encountered problems related to parsing and Abstract Syntax Trees.

Last but not least, I would like to thank Gayle Palas for her support. She was always there for me and had encouraging words when things did not go as well as planned.

1 Introduction

Many of the software systems in operation today have evolved over several years into giants with in excess of hundreds of thousands of lines of source code. Understanding and maintaining a system of such a size is laborious and expensive. Generally, system documentation is poorly maintained and hence inconsistent with respect to the original design. Many of the original designers and developers have moved on to different projects, taking their knowledge about the system with them. At the end, the only concrete and reliable piece of information about the system is the source code itself. The software engineer is then faced with extracting relevant information from this vast, unnecessarily detailed and sometimes incomprehensible amount of information.

Software is intangible with no physical shape or size and seems to disappear in files and hard drives [9]. All the information about size, complexity and structure of the system is hidden inside the files. Unlike the civil engineer who can make an immediate assessment about a structure by simply examining it visually, the software engineer is faced with searching through source code and design documents, or relying on the "Maybe..." and "I think..." answers given by the developers. It often takes a large amount of effort and time until a picture about the system is formed. It is this limited scope of visibility of the system that is broadened by the fields of software visualization and comprehension.

Software visualization uses graphical tools to display information about the system reverse engineered from the source code. Reverse engineering involves parsing the system source code and extracting programming language constructs (artifacts). Ideally, this

information is then presented in such a way that design, structural or behavioral information is recovered and emphasized, replacing or complementing existing documentation. In other words the artifacts represented are removed from the source code file level and the user can then rearrange and group bits of information which were originally spread over several source code files. Various techniques and algorithms have been devised to retrieve for example, call graphs and data dependencies to aid tasks such as subsystem identification and to measure component coupling [12]. These techniques enable the software engineer to assess the system more effectively and then plan, coordinate and monitor maintenance and reengineering tasks.

Several tools such as Rigi, Landscape, GraphLog and various others are available to visualize and analyze large software systems [6] . All of these tools follow a similar three-step strategy to generate a visual representation of the program in question:

- A data repository of programming language artifacts is created and populated with data by parsing the source code of the system.
- A querying mechanism is used to retrieve, group and organize language artifacts into abstractions.
- A flexible visualization tool allows the display and manipulation of the extracted abstractions graphically.

With respect to programs written in an object oriented language, the repository generally contains information about programming language constructs such as classes, methods and variables. Queries are designed to collect information about higher level abstractions such as class hierarchies, object composition, subsystems and so forth. Visualization then represents, for example, the class hierarchy providing a mechanism to easily manipulate the entire hierarchy at once.

1.1 The Problem

In the early 90's the concept of "Design Patterns" was introduced to the field of Computer science [3],[4].

"A Design pattern is a particular prose form of recording design information such

> that designs which have worked well in the past can be applied again in similar situations in the future" [8]

Design patterns are present all around us and subconsciously we all are aware of them. The simple concept of a door, for example, describes a general design to provide access to a room or building. Doors come in many different shapes and colors, and although only a few doors are alike we immediately recognize one when we see it. We are also aware of its structure and know about its behavior. It is remarkable how much information is associated with the word *door*. Were we presented with a door for the very first time it would take a period of time to learn its purpose, functionality, diversity and origin. However, we all are aware of the pattern and consequently all the information that is typically associated with a door. It is a classical example of a design pattern in building architecture that evolved several thousands of years ago. Architecture (truly speaking, all of nature) is full of patterns, and it seems only natural for them to emerge in the field of software design, especially as there are many similarities between software architecture and building architecture [3]. It is especially true with respect to object oriented architectures, as the object oriented paradigm is based on the idea of designing systems by modeling real world objects.

Object oriented patterns are represented by a set of classes and relationships connecting these classes. They abstract the manners in which objects are related and how they interact to solve a particular family of problems. An object oriented design pattern reflects the experiences and expertise gained by years of practicing software designers and developers. Their behavior, structure, usefulness and applicability in specific designs is well understood and documented. A large amount of information pertaining to the original systems' design, the designers intent, the problem and the solution, is hidden in the occurrence of a design pattern within the source code. It therefore makes design patterns, more precisely, identification and visualization of design patterns, primary aids to program comprehension. The advantages in providing the software engineer with information about the patterns present in a software system are considerable:

- By referencing the materials available about a pattern the software engineer can quickly learn about the design of the system, and it is possible pitfalls and advantages. Furthermore, each detected pattern immediately provides documentation about that portion of the system in which it is located.
- Parts of the system may be categorized and named in terms of the design patterns, providing a common terminology that may be used among the members of a development team. The system appears to be self-documenting.
- A software architecture or framework can be derived in terms of patterns and then compared to existing systems [14][15].

There is a pattern called "Composite", for example, that describes the necessary properties of a set of classes in order to represent a tree-like hierarchy elegantly (i.e., a recursive structure where a 'node' is composed of other 'nodes' or other 'trees'). Knowledge about the existence of a set of classes with this particular property reveals information about the structure and behavior of all the classes involved with the patterns as well as their derived classes. Information about advantages and disadvantages, performance, maintainability, scalability, etc. [2] is readily available providing the software engineer with a vast resource of information about the system on hand, without having viewed a single design document.

The visualization of object oriented design patterns within the source is an excellent tool to aid the software engineer in understanding a software system more effectively. The remainder of this thesis introduces an approach to identify and visualize the intentional or unintentional usage of object oriented design patterns within the object oriented programming language C++.

1.2 The Solution

By and large, the process of extracting and visualizing object oriented design patterns in C++ follows the strategy of building up a repository of software artifacts present in the subject system, querying the repository to reveal pattern specific structures and then visualizing the results. Only two references to similar attempts have been found so far

[6],[7]. Both attempts applied the strategy of building a repository and then querying the latter for patterns. Major differences between this work and either one of [6],[7] lie in the complexity of the software artifacts extracted and the methods used to identify design patterns within the repository.

Although the authors of [6] composed a far more elaborate repository than introduced in this thesis, their queries were mainly based on matching the names of classes extracted from the source code with the names of patterns introduced in [6]. This method will only find patterns in which the designer insisted on using the nomenclature of the pattern. Moreover, incidental use of names may result in false positive identifications.

This work is based on Kreamer's work [7], who composed a data repository by extracting artifacts from C++ header files and then stored these artifacts in the form of Prolog language facts. His idea has been adopted and further developed to include artifacts extracted from the C++ definition files, and to provide a more elaborate querying mechanism also based on the Prolog language. Although much of the information extracted from the header files is necessary to identify design patterns, it is not sufficient. Part of the patterns' structure involves information that can only be extracted from the definition files. As a result Kraemer's queries contained many free variables that resulted in false positive identifications.

Emphasis is put on finding design patterns that conform very closely to the ones introduced by Gamma *et al.* [2]. Unfortunately, due to the nature of patterns making them architectural examples rather than concrete implementations, variations of such are frequent and may hence be missed.

1.2.1 The Data Repository

Before extracting object oriented design patterns from source code one has to first identify and describe what exactly defines a design pattern. What are its constructs? What does it look like? Next, these constructs have to be extracted from the source code, formatted and consolidated into a repository. The repository's purpose is to provide easy access to manipulate, select and group the constructs and to avoid frequent re-parsing of the same

piece of source code [22].

In "Design Patterns: Elements of Reusable Object-Oriented Software" [2] the authors describe a variety of object oriented patterns using the terminology and constructs inherent to object oriented design and programming. These constructs, also referred to as *Object Modeling Technique* (OMT) constructs, include classes, inheritance, member functions, call delegations and so forth. The design patterns are described using the OMT formalism. It is adapted here, defining to a large extent the software artifacts that are extracted in order to identify the design patterns.

The constructs are either directly extracted or easily computed from the Abstract Syntax Tree (AST) representation of the program in question [11]. ASTs are created by parsing the source code and adding programming language constructs to the tree. Compilers, for example, create ASTs to decompose the program into its individual constructs to simplify semantic analysis and translation into machine code. The programmable public domain C++ parser *cppp*, publicized by Brown University, was the tool chosen to aid in the extraction of the AST from C++ source code.

By traversing the AST, programming language artifacts are identified and extracted. The artifacts are then recorded in the data repository in the form of *Prolog language facts*. The facts are initially stored in text files, sometimes referred to as the *fact database*. From there they are read in by the *Prolog runtime environment*, where queries in the form of *Prolog clauses* are executed. The collection of queries is also part of the data repository and referred to as the *clause database*.

1.2.2 Pattern Queries

The identification of patterns among the data extracted from the source code is done by applying various Prolog clauses to the repository. The clauses list all the software artifacts that are required to compose a design pattern. Once a clause has been satisfied, i.e., all the artifacts listed in the clauses are present in the repository, an occurrence of a pattern has been detected and a corresponding fact is added to the repository. Further processing is done to remove duplicates, consolidate the results and produce as output a small report.

1.2.3 Visualization Techniques

The visualization tool uses the data generated during querying to represent the patterns visually. The source code is displayed as a graph, where classes are nodes and arcs represent relationships among classes. Relationships include inheritance, reference, call delegations and so forth. The visualization technique used here displays the individual patterns within the class hierarchy. Classes composing a pattern are colored depending on their position within the pattern.

1.3 Summary

Object oriented design patterns encapsulate expertise gained by practitioners throughout the field. A design pattern contains more information about system design, structure and behavior than the individual software abstractions of which it is composed. Identification and visualization of intentional or unintentional use of patterns within the source code of a system is a valuable aid to the software engineer in maintaining and/or re-engineering the system. The process of pattern identification involves the strategies of reverse engineering and querying, already applied throughout the fields of software engineering and visualization.

2 Object Oriented Design Patterns

The term design pattern was first introduced in the 1960's by Christopher Alexander, an architect, who researched automated design and modular construction [21]. He found recurring themes in architecture which he referred to as design patterns. The term has now been adopted in many other disciplines ranging from design patterns in business management to computer programming and design.

Generally speaking, patterns in design are used to standardize the way designs are developed. The remainder of this chapter is an introduction to patterns related to object oriented programming and design. Several patterns relevant to the remainder of this thesis are introduced.

2.1 Object Oriented Design Patterns

As software developers struggle to design software systems in a maintainable, extensible and scalable fashion, many are faced with similar design and implementation specific problems. Naturally, through the iterative process of implementation and redesign many also arrive at similar solutions. It is the commonalities or patterns among these solutions that is referred to as a design pattern. A design pattern describes a solution, or a framework for a solution, to recurring problems in software design.

A design pattern consists of a name, a problem statement, a context in which the problem may occur and finally a description of a solution to the problem [4]. The solution

is given by combining various programming language constructs and abstractions to form a yet higher level of abstraction — the design pattern. The structure of a pattern is specified diagrammatically using the Object Modeling Technique (OMT) inherent to object oriented design and programming.

Design patterns have received much attention lately and a great effort is currently under way in academia as well as industry to identify, document and classify various software design patterns. They are powerful tools for software designers as each pattern encapsulates design expertise gained by practitioners throughout the field. A software architect knowledgeable of proven design structures can immediately apply them to a problem without rediscovering them [1].

Gamma *et al.* suggest classification of object oriented design patterns into three separate families related to object creation and destruction, structure, and behavior [2].

2.1.1 Creational Design Patterns

Creational object oriented design patterns abstract the class instantiation processes [2]. They hide information about how, where and when an object is created while maintaining a constant interface to the desired object. As object oriented programming and design moves away from inheritance and puts more emphasis on object composition, objects tend to be composed to implement specific behaviors at runtime. A simple instantiation is often not enough and what requires more elaborate means to create an object. Various creation strategies are captured in creational patterns.

2.1.2 Structural Design Patterns

Structural patterns are concerned with how classes and objects are composed to form larger structures. They use inheritance and object composition to provide flexible and modifiable interfaces or implementations [2].

2.1.3 Behavioral Design Patterns

Behavioral patterns are concerned with algorithms and the assessment of responsibilities between objects. They describe the flow of control and communications between objects. Call sequences and timing issues are addressed as well as object compositions and structures.

2.2 Object Modeling Techniques

The Object Modelling Technique (OMT) is used to document and specify object oriented designs [2]. Object models are described by combining classes and the relationships connecting these classes such as inheritance, references and call delegations into higher levels of abstraction. A design pattern is little more than an object oriented model, and it can easily be represented using OMT. Introduced are the concepts underlying various OMT constructs which are then used to specify individual design patterns. Only the OMT constructs applied in the remainder of this work are introduced.

Although most object oriented design patterns can be modeled using OMT, in the case of behavioral patterns these constructs may not be sufficient. Behavioral patterns also describe runtime properties such as call sequences and object lifetimes. Charts similar to message sequence charts are used to describe time dependent behavior. Within this thesis, however, only the constructs required to specify the structural aspects of design patterns are presented and discussed. None of the patterns introduced display any complex runtime behaviors.

2.2.1 Classes

A class defines an object's internal data and member functions. Generally there are several classes in a pattern, related by either inheritance or object composition. Classes are categorized as either abstract or concrete which is of great importance for the structure and

intent of the pattern. Abstract classes are generally used to define interfaces, by defining a set of virtual functions which either implement a default behavior, or are not implemented at all. Derived classes then implement the interface defined by the abstract class.

The classical definition of an abstract class in C++ is too strict and has been relaxed here. A class is considered to be abstract if it contains at least one virtual function and not as in C++, one *pure* virtual function. Other than the fact that an abstract class in C++ cannot be instantiated the concept applied here is the same. It is of importance that an extensible interface is specified by an abstract class. Whether it is fully implemented or not is irrelevant.

Diagrammatically, classes are represented by a box with the class name in bold as the first line of text, followed by optional member functions and variables. Italicized class names and member function names represent abstract classes and virtual functions respectively. The member functions may be annotated by pseudo code, displayed in a separate text box that contains a solid square in the upper right corner.

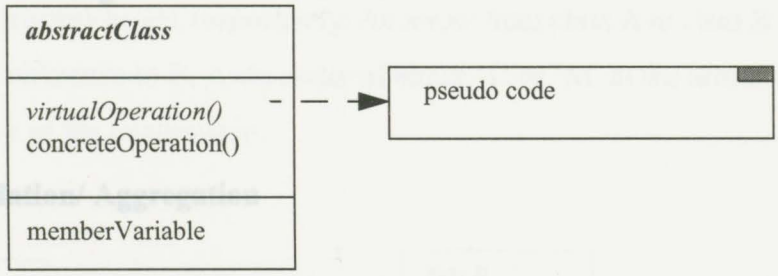

Figure 2.1 Class

2.2.2 Inheritance

Inheritance provides a means of creating new classes by combining and extending existing classes. It is one of the key concepts of object oriented design and programming. If a class A is inherited from a class B then A is called a *subclass* of B and B is called the *superclass* or base class of A. One usage of inheritance is to extend the functionality of a class by

inheriting from it and then adding more operations. The new class provides all the operations of the base class as well as the additional ones. Inheritance is also used to provide constant interfaces to a series of classes by first creating an abstract class that defines an interface using virtual functions. Then the functions are overridden and re-implemented in the subclasses to provide various implementations to the interface.

Inheritance is represented by a line connecting two classes with an arrow pointing from the subclass to the superclass.

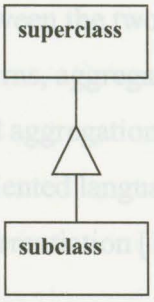

Figure 2.2 Inheritance

2.2.3 Association/ Aggregation

Association and aggregation describe the ways one object maintains a reference to another object. An object A is said to associate to object B if it maintains a pointer or reference to B. An object X is said to aggregate an object Y if Y is a member of X. In other words class Y is declared to be a member variable of class X, implying that the two objects have the same lifetime and furthermore making X responsible for creating and destroying Y. Associations and aggregations are further associated with a cardinality, which indicates whether one or many objects are referenced.

Along with inheritance, references are also used as a means to compose objects. Combined with call delegation object references can be used to achieve effects similar to

inheritance. Consider, for example, a class that implements its own interface by calling the operations of an object it references. The actual implementation of the interface is delegated to another object and various implementations can be achieved by referencing different objects. This strategy is similar to defining an abstract class and subclassing from it to provide various implementations to the interface. Recently, some in the object oriented community have suggested moving away from the latter approach to make more use of object composition [2]. It reduces the number of different classes and provides looser coupling between interfaces and implementations.

Although an implicit distinction between the two methods of object reference are made in the specification of design patterns, aggregation and association may simply be regarded as a reference. Association and aggregation can often be implemented either way, or, as is the case with the object oriented language Java, there are no language constructs that allow for that kind of differentiation [18]. Only in a few specific cases is the distinction of importance.

Diagrammatically, associations and aggregations are represented by arrows with solid and transparent arrow heads, respectively. An arrow from class A to class B represents that A maintains a reference to B. A character of either '1' or 'M' at the arrow head indicates the multiplicity of the relationship.

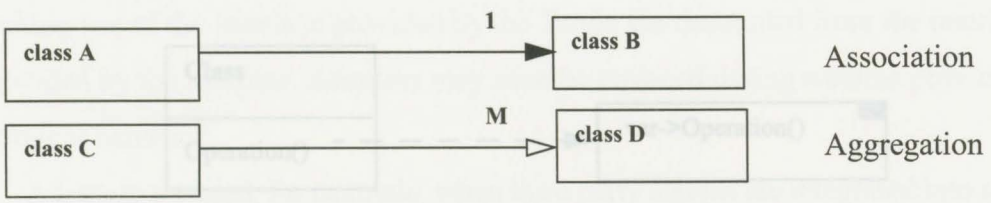

Figure 2.3 Association and Aggregation

2.2.4 Calls and Delegation

Call and delegation relationships represent one member function calling another. The relationships are indicated as pseudo code annotating a particular member function. Although the representation of call and delegation relationships are identical, their intent differs considerably. A delegation occurs when one object takes a request and simply passes or delegates that request to another object, by calling one of that object's member functions. It is a technique commonly used in object composition and is similar to superclasses deferring implementation of certain functions to sub classes. Here the implementation is deferred to another object. Delegation implies that the function which delegates the call is designed for explicitly that purpose and contains no other function calls. Therefore a function is considered to delegate if and only if it issues a single function call. A call relation on the other hand, simply refers to the one function calling another. A function that issues several functions calls maintains call relations to these functions but does not delegate.

Calls and call delegations are represented by pseudo code annotating the object member functions. Which of the two is intended is taken out of the context in which it is used.

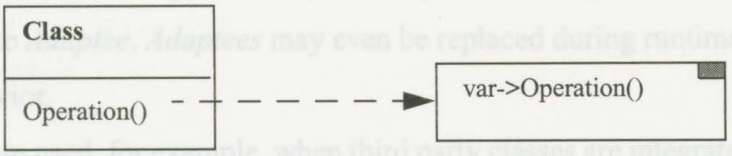

Figure 2.4 Call and Delegation

2.3 Selected Patterns

This section discusses several selected object oriented design patterns used in the remainder of this thesis. The intent and usage of each pattern is discussed and an OMT representation is provided. The discussion of the patterns is kept to a minimum. Please refer to Gamma *et al.* for a more detailed presentation of the individual patterns [2], [1]. Due the fact that design patterns are regarded as architectural examples or guidelines rather than concrete implementation specific layouts, their representation may leave much room for varying interpretation and implementation. Although the patterns discussed in this section closely conform to the representations in [2], the subsequent chapters introduce slight variations. Three structural pattern and one behavioral pattern are described.

2.3.1 The Object Adapter

The Adapter pattern, probably the simplest of the structural patterns, is used to convert the interface of a class to one that a *Client* class expects. It allows classes to work together that otherwise could not, because of incompatible interfaces. An *Adapter* implements the interface inherited from the *Target* by delegating function calls to the *Adaptee*. *Clients* making use of the interface provided by the *Target* are decoupled from the interface provided by the *Adaptee*. *Adaptees* may even be replaced during runtime providing a different behavior.

Adapters are used, for example, when third party classes are integrated into an existing system. Consider the case where a window class expects the drawable objects it contains to provide a function `draw()`, used to refresh the screen. In order to introduce a new drawable object that, for example, provides a function called `paint()` instead of `draw()` an Adapter is created that adapts the new drawable object to the interface expected by the window class.

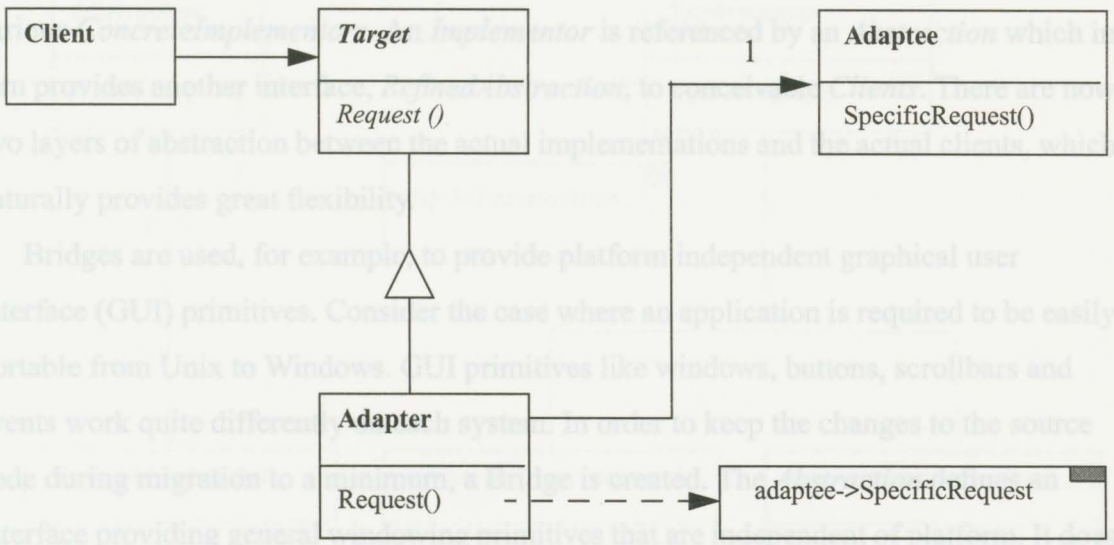

Figure 2.5 Object Adapter pattern structure

- The *Target* defines the interface expected by the *Client*.
- The *Adapter* adapts the interface of the *Adaptee* to the one inherited from the *Target*.
- The *Adaptee* is the class whose interface is to be converted.
- The **Client** makes use of the interface provided by the **Target**.

2.3.2 The Bridge

The Bridge pattern decouples the abstraction from its implementation so that the two can vary independently. It is used when an abstraction can have one of several possible implementations. The common solution to this problem is to define the interface via an abstract class and use subclasses to implement the interface in various ways. However, using inheritance to provide the implementation tightly couples the interface and its implementation making it difficult to modify, extend or use independently.

The Bridge pattern avoids this problem by associating the implementation rather than

inheriting it and by adding another layer of abstraction on the implementations side as shown in Figure 2.5. The abstract class *Implementor* provides an interface implemented by various *ConcreteImplementors*. An *Implementor* is referenced by an *Abstraction* which in turn provides another interface, *RefinedAbstraction*, to conceivable *Clients*. There are now two layers of abstraction between the actual implementations and the actual clients, which naturally provides great flexibility.

Bridges are used, for example, to provide platform independent graphical user interface (GUI) primitives. Consider the case where an application is required to be easily portable from Unix to Windows. GUI primitives like windows, buttons, scrollbars and events work quite differently on each system. In order to keep the changes to the source code during migration to a minimum, a Bridge is created. The *Abstraction* defines an interface providing general windowing primitives that are independent of platform. It does so by making use of the interface provided by the *Implementor* it references. These primitives are then used throughout the application source code. The *Implementor's* interface is implemented by a derived *ConcreteImplementor* that interface to individual platforms by issuing the appropriate system calls. A *ConcreteImplementor* is provided for each conceivable platform on which the system may run. It is then implemented in such a way that the platform is determined at runtime. Depending on the operating system, the *Abstraction* is initially created referencing the appropriate *Implementor*. No recompilation is necessary. The programming language Java, for example, makes use of Bridge patterns to provide platform independence as a part of the language [18].

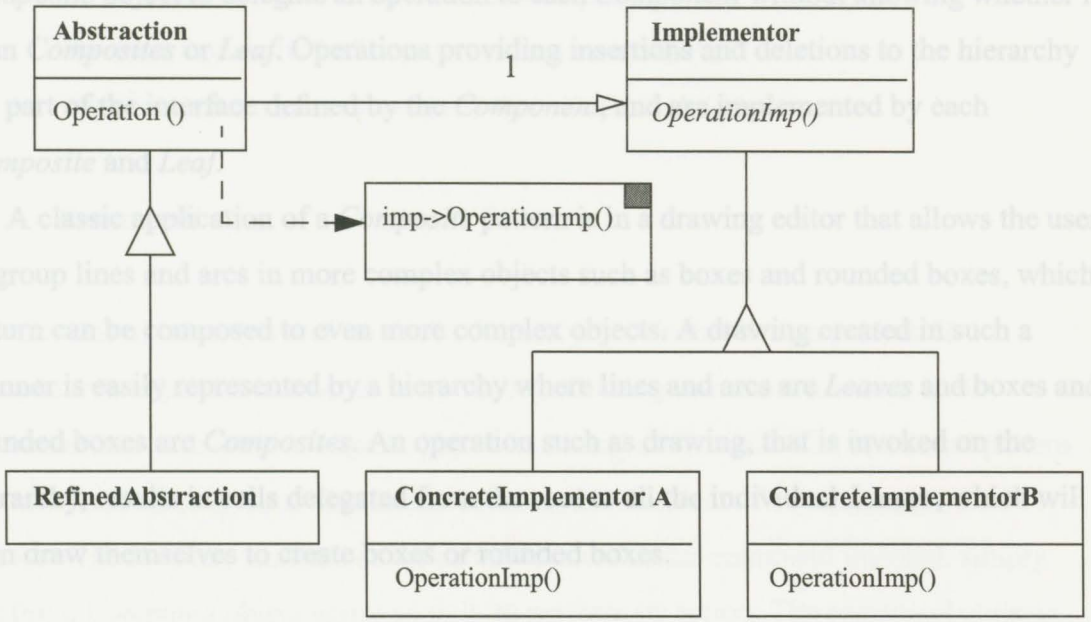

Figure 2.6 Bridge pattern structure

- The *Implementor* provides a constant interface to various *ConcreteImplementors*.
- The *ConcreteImplementor* implements the operations provided by the *Implementor*'s interface.
- The *Abstraction* associates an *Implementor* and uses it to provide a higher level interface to a *Client*.
- The *RefinedAbstraction* extends the interface provided by the *Abstraction*.

2.3.3 Composite

The Composite is probably the most intriguing of the structural patterns. It describes a recursive structure that can be used to compose objects into tree-like hierarchies. An object within the structure is either a simple leaf or a composite. A *Leaf* is a final node in the hierarchy and a *Composite* is a node consisting of a collection of either type. The pattern's main emphasis is on making the interface of each *Leaf* and *Composite* conform

to a single interface provided by their common base class *Component*. This allows a *Composite* object to delegate an operation to each *Component* without knowing whether it is an *Composites* or *Leaf*. Operations providing insertions and deletions to the hierarchy are part of the interface defined by the *Component*, and are implemented by each *Composite* and *Leaf*.

A classic application of a *Composite* pattern is in a drawing editor that allows the user to group lines and arcs in more complex objects such as boxes and rounded boxes, which in turn can be composed to even more complex objects. A drawing created in such a manner is easily represented by a hierarchy where lines and arcs are *Leaves* and boxes and rounded boxes are *Composites*. An operation such as drawing, that is invoked on the hierarchy, results in calls delegated from the root to all the individual *Leaves*, which will then draw themselves to create boxes or rounded boxes.

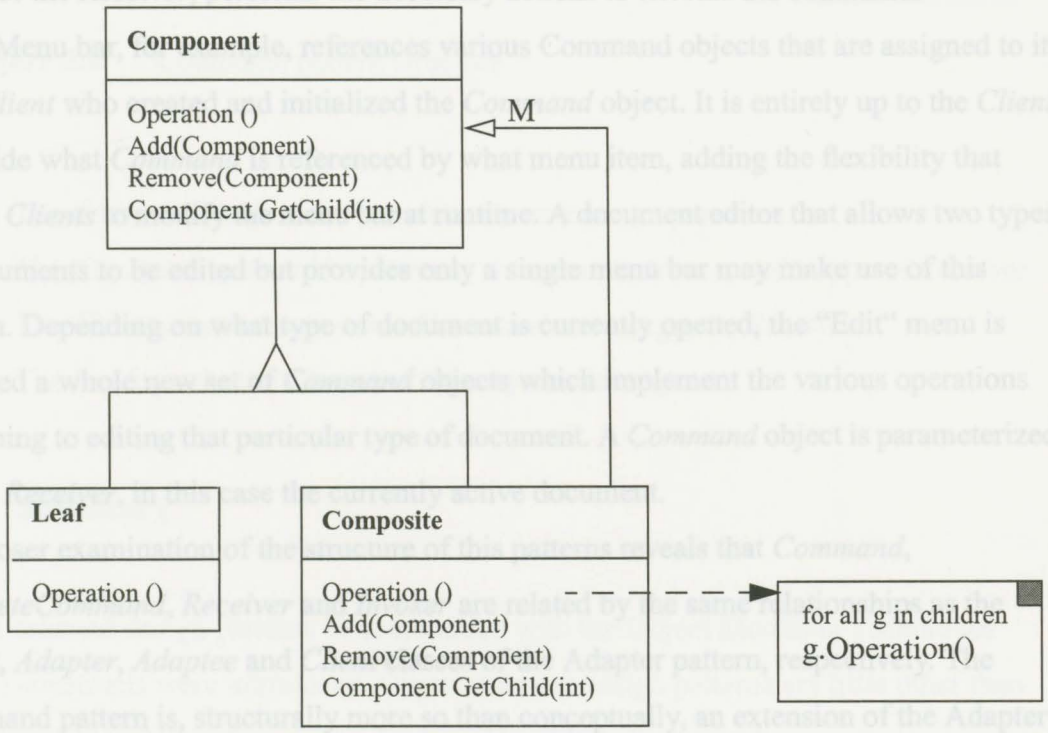

Figure 2.7 Composite pattern structure

- The *Component* defines the interface for objects in the composition and implements default behaviors for operations if possible.
- The *Leaf* represents leaf objects and has no children.
- The *Composite* represents objects that can have children.

2.3.4 Command

The Command is categorized as a behavioral object oriented design pattern. It encapsulates requests/commands as an object, allowing command invokers to be parameterized with various requests, by maintaining a reference. The command pattern introduces another level of abstraction between the *Receiver* of a command and its *Invoker*. The *Invoker*, unaware of the actual receiver of the command invoked, simply notifies a *Command* object assigned to it, to perform its action. The command object, aware of the *Receiver*, performs the necessary actions to execute the command.

A Menu bar, for example, references various Command objects that are assigned to it by a *Client* who created and initialized the *Command* object. It is entirely up to the *Client* to decide what *Command* is referenced by what menu item, adding the flexibility that allows *Clients* to modify the menu bar at runtime. A document editor that allows two types of documents to be edited but provides only a single menu bar may make use of this pattern. Depending on what type of document is currently opened, the "Edit" menu is assigned a whole new set of *Command* objects which implement the various operations pertaining to editing that particular type of document. A *Command* object is parameterized with a *Receiver*, in this case the currently active document.

Closer examination of the structure of this patterns reveals that *Command*, *ConcreteCommand*, *Receiver* and *Invoker* are related by the same relationships as the *Target*, *Adapter*, *Adaptee* and *Client* classes of the Adapter pattern, respectively. The Command pattern is, structurally more so than conceptually, an extension of the Adapter pattern.

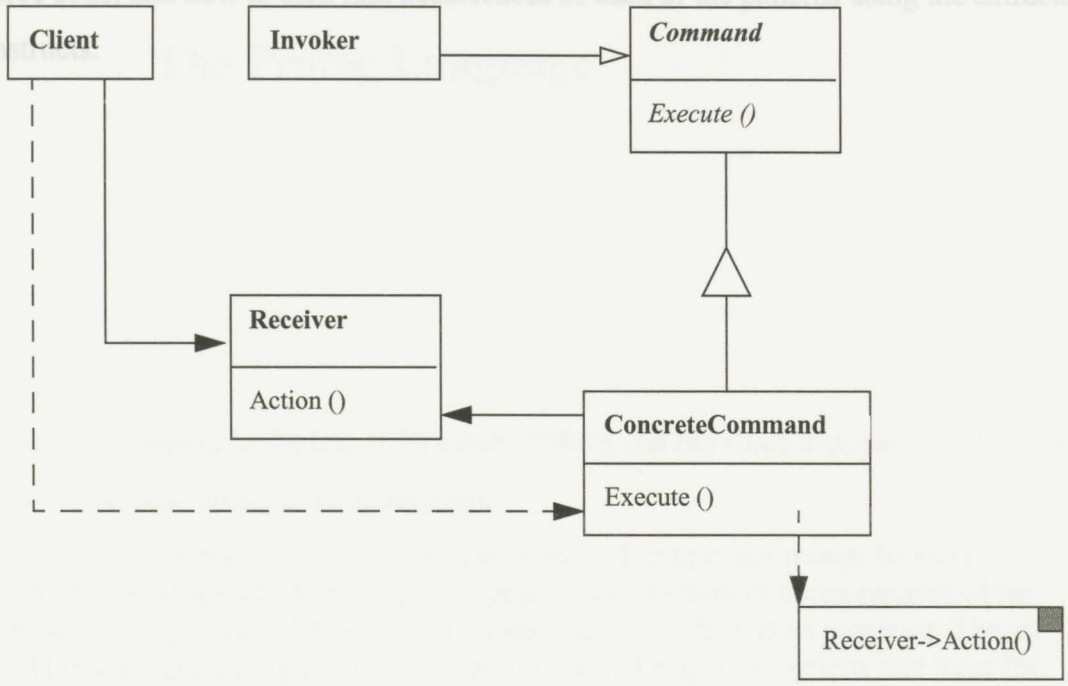

Figure 2.8 Command pattern structure

- The *Command* declares an interface for executing operations.
- The *ConcreteCommand* implements an operation for a specific type of *Receiver.*
- The *Client* creates the *ConcreteCommand* and assigns its *Receiver.*
- The *Invoker* asks the *Command* to carry out the request.

2.4 Summary

Object oriented design patterns in conjunction with the Object Modeling Techniques (OMT) constructs were introduced. Object oriented design patterns are little other than object models and therefore can be described by OMT diagrams. Four patterns named *Adapter*, *Bridge*, *Command* and *Composite* were introduced and their application discussed.

Following chapters will present how to reverse engineer OMT constructs from C++ source code, and how to then find occurrences of each of the patterns using the extracted constructs.

3 The Prolog Language

Prolog was invented in the late 1970's early 1980's and has since then become the most widely used logic programming language.

> "A logic program is a set of axioms, or rules, defining relationships between objects. A computation of a logic program is a deduction of consequences of the program. A program defines a set of consequences, which is its meaning. The art of logic programming is constructing concise and elegant programs that have the desired meaning"[10].

This chapter describes the features of the Prolog language which are applied and referred to throughout the remainder of this thesis. Emphasis is put on showing how Prolog stores information about an environment and how it provides an effective and easy to use means to detect the presence of relationships within that environment.

3.1 Execution

Execution of a Prolog program is usually done via a Prolog interpreter. The interpreter is a command line based runtime environment into which clauses and facts are either entered interactively or read from a file. The information entered is stored in a database maintained by the runtime environment. Once clauses and facts are declared, a goal may be entered which the interpreter attempts to satisfy, given the facts and clauses in the internal database.

The Prolog runtime environment provides basic arithmetic and I/O primitives as well

as an interactive debugger.

3.2 Facts

Within Prolog, facts are a means of stating that a relationship holds among objects [10]. Generally facts are used to represent pieces of data and their properties that are available about an environment. A fact consists of an identifier and any number of atoms separated by commas. Each atom encapsulates a piece of information pertaining to the fact. For example,

```
father (tom, kim).
mother (judy, kim).
```

are two facts stating that `tom` is a `father` of `kim` and `judy` is a `mother` of `kim`. `Father` and `mother` are referred to as the facts and `tom`, `judy` and `kim` as atoms. Here, facts as well as atoms start with lower case letters. In Prolog, identifiers starting with upper case characters are interpreted as variables. For example,

```
sister (X, Y).
```

means that everyone is a `sister` of everyone. Variables are rarely used with simple facts and occur more frequently when specifying clauses and executing goals.

3.3 Rules and Clauses

Rules or clauses define relationships among facts or other clauses. In other words, rules can be used to define more elaborate relationships among the facts in the internal database. Generally clauses define relationships among the pieces of data in an environment. For example,

```
hasParents(A) :- father(X, A), mother(Y,A).
```

describes a Prolog clause. The clause `hasParents (A)` states that atom `A` has parents if

there exists an atom X such that `father(X, A)` is satisfied and if there exists a Y such that `mother(Y, A)` is satisfied as well. `Father(X,A)` and `mother(X,A)` are referred to as terms of the clause `hasParents(A)`. The comma separating the terms indicates that they are related by a logical 'and' relation. Although there are various logical operators, the clauses introduced in the remainder of this thesis are restricted to the use of the comma ('and') and semicolon ('or') logical operators.

Clauses model and subsequently detect relationships among pieces of data presented as facts. Clauses can be nested allowing an incremental approach to design complex relationship models.

3.4 Goal

A goal is a means of querying for the existence of atoms that satisfy facts or clauses. It is entered directly into the Prolog interpreter, which then attempts to satisfy the goal. For example,

```
father (tom, kim) ?
```

states a goal that is satisfied if the fact `father(tom, kim)` is present in the interpreters' internal database. In this case, after the goal has been entered, the Prolog interpreter responds with

```
yes.
```

indicating that the fact is present. Goals can also be paramaterized with variables which causes the interpreter to find all those atoms that satisfy the goal. For example,

```
father(X, kim) ?
```

is satisfied if there exists an atom that instantiates X so that the fact `father(X,kim)` exists. The Prolog interpreter answers the clause with all possible instantiations of X that satisfy the goal.

Clauses can be entered as goals as well. The interpreter attempts to satisfy all the terms present in the clause. Once that has occurred the clause has been satisfied as well and the goal is achieved. For example,

```
hasParents(kim) ?
```

shows how a clause is executed. Here we have instantiated the variable A with the atom `kim`. The interpreter now attempts to instantiate `X` and `Y` by satisfying each of `father(X,kim)` and `mother(X,kim)` independently. Then the logical operator AND is applied to the results. If both subclauses can be satisfied then the clause `hasParents(kim)` is satisfied as well. Similarly,

```
hasParents(A)
```

lists all possible atoms A for which the clause holds. Depending on the size of a clause and number of variables involved, the process of satisfying the goal may be extremely runtime intensive. The Prolog interpreter repeatedly traverses the internal database in search of an instantiation of variables that satisfy the clause.

3.5 Variables

Generally variables are used to parameterize goals, causing the Prolog interpreter to find all possible instantiations of that variable which satisfy the goal. However, at times variables are also present within facts and clauses, acting as general place holders that are instantiated by the interpreter whenever possible. As a general rule, a goal is never satisfied unless all variables are instantiated. A goal that contains several variables, for example,

```
father (X,Y) ?
```

produces as its output all possible tuples of `(X,Y)` that satisfy the rule. Now suppose one only wanted to know who the fathers are, but does not care about who the children are. In other words we want the interpreter to only report the value of one of the variables. Prolog

provides the special variable underscore (_) for just this case. For example

```
father(X, _) ?
```

produces as its output all possible values for X such that X is anyone's father . Although the underscore is instantiated by the interpreter it is not output. Generally speaking, the underscore provides a means of limiting the output produced without actually limiting the query.

3.6 Summary

This short introduction to Prolog showed how it can be used to represent and store various aspects of an environment in terms of a language construct called a *fact*. A fact can contain a large number of individual pieces of information called *atoms* which combined uniquely identify the fact. Among many other applications Prolog is ideally suited to represent artifacts extracted from an object oriented program. A function, for example, can be represented by a fact named "function" whose atoms are the name, class of which it is a member, access specifier and so forth.

Prolog also provides powerful means of extracting relationships among a set of facts. Clauses modeling these relationships can be specified by conjunctively or disjunctively listing the required relationships. Once a clause is executed it will not only confirm the presence of a set of relationships but also present the user with a the pieces of information that define a relationship. Subsequent chapters will show how Prolog can be used to define and execute object oriented pattern queries.

4 System Architecture

In this chapter we describe the architecture and functionality of the system devised to extract, identify and visualize object oriented design patterns. This system is designed to implement the strategies of reverse engineering, querying and visualization described in chapter 1, by applying and combining various proven methodologies and readily available software and tools. Techniques and expertise from the following fields of Computer Science are exploited, and are reflected within the individual components of the architecture:

- Logic programming
- Source code parsing and compiler technology
- Software visualization
- Object oriented design
- Software engineering

Use of the third party sources greatly simplified the task of devising and implementing the introduced system, resulting in a robust and easily reproducible architecture. Standing on a giant's shoulders truly proved to provide a better view.

4.1 Conceptual System Overview

This section describes an conceptual overview of the tasks involved in the pattern detection process. The process is divided into three separate tasks concerning pattern query design, source code pre-processing, and querying and visualization. Figure 4.1 presents an overview of the steps performed within each task.

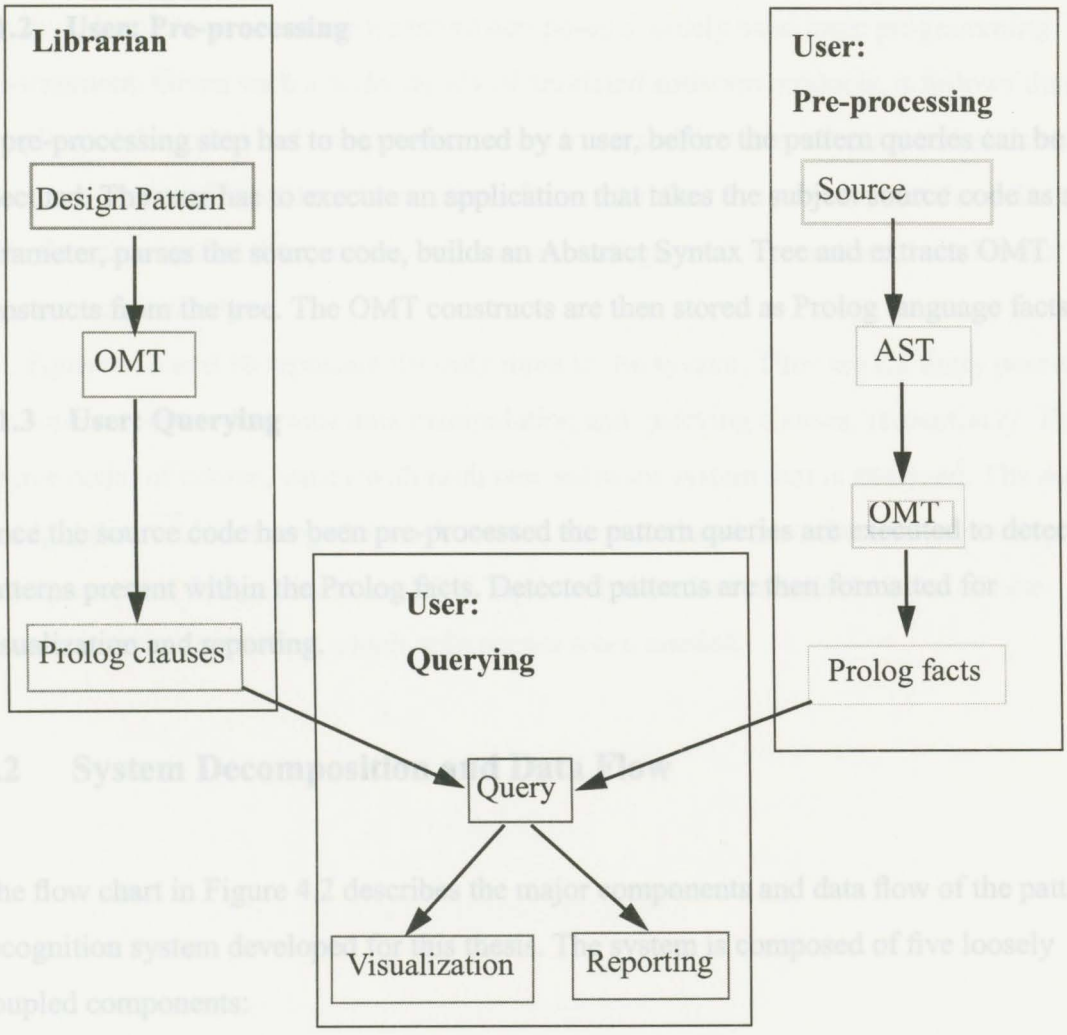

Figure 4.1 Conceptual System Overview

4.1.1 Librarian

A librarian is responsible to provide a set of design pattern prolog clauses that can be used by the user to detect and process individual patterns. The clauses are created by translating the pattern's OMT diagram. Once the clauses have been created they can be applied to a number of subject systems.

4.1.2 User: Pre-processing

A pre-processing step has to be performed by a user, before the pattern queries can be exectued. The user has to execute an application that takes the subject source code as a parameter, parses the source code, builds an Abstract Syntax Tree and extracts OMT constructs from the tree. The OMT constructs are then stored as Prolog language facts.

4.1.3 User: Querying

Once the source code has been pre-processed the pattern queries are executed to detect patterns present within the Prolog facts. Detected patterns are then formatted for visualization and reporting.

4.2 System Decomposition and Data Flow

The flow chart in Figure 4.2 describes the major components and data flow of the pattern recognition system developed for this thesis. The system is composed of five loosely coupled components:

- Source code parser.
- Prolog fact generator.
- Data repository.
- Prolog runtime environment.

- Visualization tool (Rigi).

The source code parser, fact generator and data repository perform the reverse engineering task. The Prolog runtime environment acts as a querying engine, and Rigi provides the visualization of the results. Executable components in the system are the parser, fact generator, Prolog runtime environment and visualization tool. The data repository manages a collection of data gathered throughout various stages of the process.

The parser, and visualization tool are off-the-shelf third party applications, and Prolog combined with its runtime environment composes a widely used logic programming environment. Given such a wide variety of unrelated software products, it follows that coupling of the individual components is very loose and that execution as well as data flow is not automated. The order of execution of the individual components and the order of data flow throughout the system is represented in Figure 4.2 by numbered circles and squares, respectively.

Squares 1a and 1b represent the only input to the system. They are the entry points for the source code, and various data manipulation and querying clauses, respectively. The source code, of course, varies with each new software system that is analyzed. The data manipulation and querying clauses, however, are reused and the collection grows larger as new queries are designed and modeled. Square 1b therefore represents input of new clauses into the repository, which only occurs when needed.

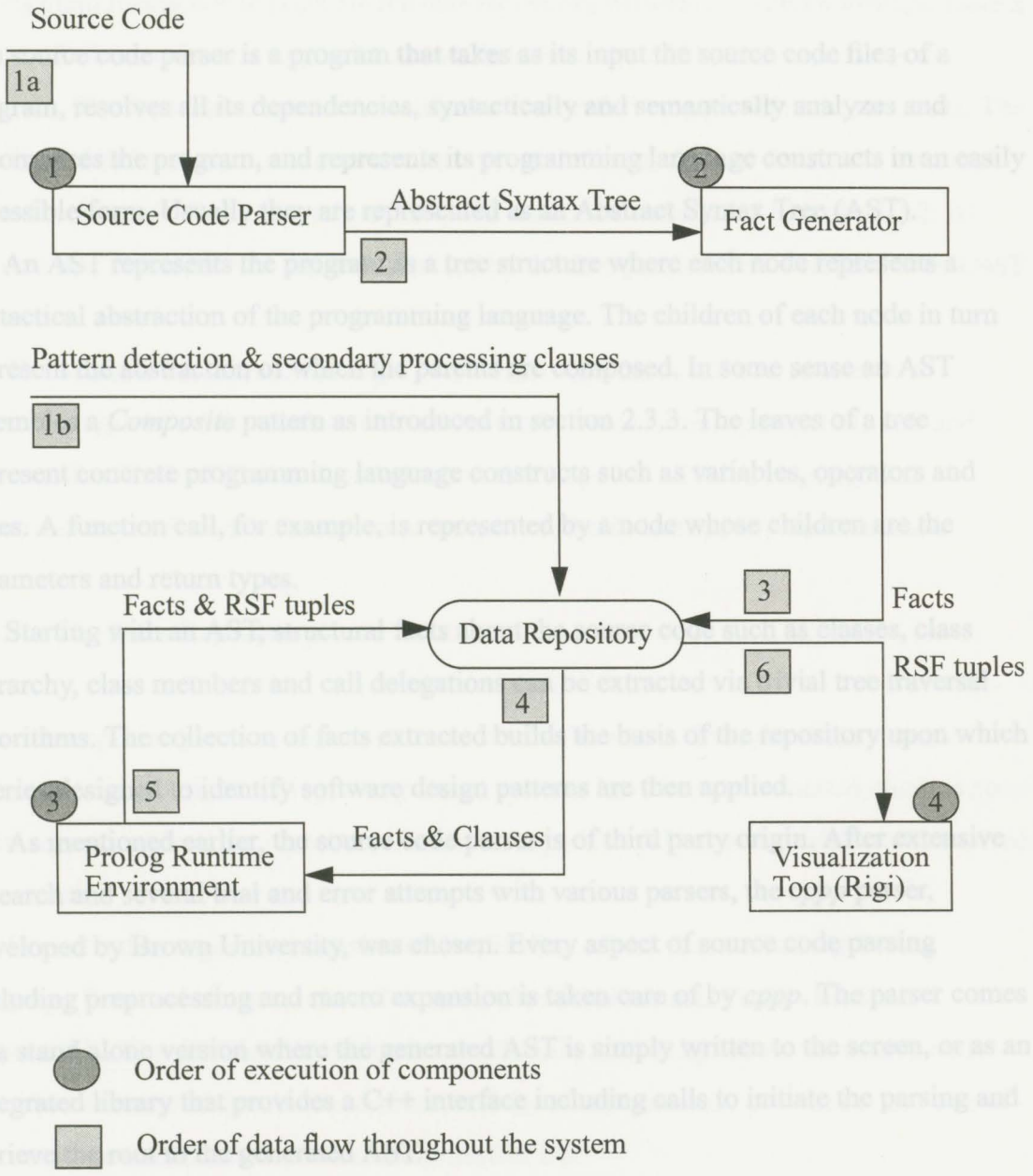

Figure 4.2 Architecture of pattern recognition system

4.2.1 The Source Code Parser

The source code parser is a program that takes as its input the source code files of a program, resolves all its dependencies, syntactically and semantically analyzes and decomposes the program, and represents its programming language constructs in an easily accessible form. Usually they are represented as an Abstract Syntax Tree (AST).

An AST represents the program as a tree structure where each node represents a syntactical abstraction of the programming language. The children of each node in turn represent the abstraction of which the parents are composed. In some sense an AST resembles a *Composite* pattern as introduced in section 2.3.3. The leaves of a tree represent concrete programming language constructs such as variables, operators and types. A function call, for example, is represented by a node whose children are the parameters and return types.

Starting with an AST, structural facts about the source code such as classes, class hierarchy, class members and call delegations can be extracted via trivial tree traversal algorithms. The collection of facts extracted builds the basis of the repository upon which queries designed to identify software design patterns are then applied.

As mentioned earlier, the source code parser is of third party origin. After extensive research and several trial and error attempts with various parsers, the *cppp* parser, developed by Brown University, was chosen. Every aspect of source code parsing including preprocessing and macro expansion is taken care of by *cppp*. The parser comes in a stand alone version where the generated AST is simply written to the screen, or as an integrated library that provides a C++ interface including calls to initiate the parsing and retrieve the root to the generated AST.

4.2.2 The Fact Generator

The Fact Generator is the only custom-made component of the system. It is a C++ application that requires as its input a pointer to the root of an AST. Using the *cppp* library's C++ interface, the Fact Generator initially invokes the parser to obtain an AST of

the source code on hand.

Its main function is to populate the data repository with the domain knowledge about a system. It does so by deriving software artifacts from the AST which represents the program. The type and extent of the artifacts are specified by a predefined data model. The data model defines what information to extract and what to discard during the traversal. Here the data model corresponds to the OMT constructs introduced in section 2.2. It follows that each software artifact extracted by the fact generator also resembles an OMT construct.

The artifacts are extracted using tree traversal algorithms tailored to extract a particular type of artifact such as class, inheritance, association, aggregation, call and instantiation. Each artifact is then formatted and output as a Prolog language fact, encapsulating sufficient information to uniquely identify and specify the construct.

4.2.3 The Data Repository

The data repository acts as the central source of information throughout the system. Although many variants of repository-based systems have been constructed, there is no clear schema on how the repository should be organized [22] or of the nature of the stored data entities.

Here the data repository consists of a collection of text files containing Prolog language facts and clauses, referred to as the *fact database* and *clause database*, respectively. The *fact database* contains the domain knowledge about the source code specified in terms of data model entities, and the *clause database* provides a tool box of queries, data manipulation means and primitive I/O.

Initially, the *fact database* is populated with the domain knowledge about the system, represented by all information about the source code extracted by the Fact Generator. It contains a static basis for all subsequent queries to avoid re-parsing of the system.

The *clauses database* represents an ever growing repository of queries as well as formatting and I/O clauses. Individual pattern queries are added to the repository when needed and will then always be present and available.

In addition to Prolog language facts the data repository contains a file of Rigi Standard Format (RSF) tuples. RSF is the input format for Rigi, which is the chosen visualization tool. RSF is generated by dedicated clauses that format and output the contents of the *fact database* as RSF tuples.

The Prolog language statements are read by the Prolog runtime environment, where they are stored internally. Once present in the environment the clauses are executed to query for, consolidate and output individual patterns. The RSF file is read by the visualization tool which then allows the user to examine and rearrange the extracted source code and pattern artifacts.

4.2.4 The Prolog Runtime Environment

Sicstus V2.0 was chosen as the Prolog runtime environment. Sicstus is a command line based Prolog interpreter that allows the user to enter Prolog facts and clauses either directly or from a file. Once a clause has been entered into the environment it can be run in an attempt to satisfy it. Satisfying a clause that models a pattern means that the pattern has been detected within the facts presented to the interpreter. As the facts are directly derived from the source code, it follows that the pattern is present there as well. The queries and facts are designed in such a way that a detection of a pattern also provides detailed information about the classes involved. This information is then added to the RSF file.

4.2.5 The Visualization Tool (Rigi)

Rigi provides the flexibility to view and arrange the retrieved patterns among the language artifacts extracted from the source code. The visualization is based on a directed graph, where nodes and arcs represent various software artifacts. The user can rearrange, filter and annotate various aspects of the graph providing analysis beyond the possibilities of pure textual means.

Although Rigi provides its own reverse engineering functionality, including parsers for various languages, only its visualization functionality is utilized here. Input for the tool

has been derived from the data repository, rather than having Rigi parse the source code and generate its own repository. In addition to generating input for Rigi, a Rigi domain had to be defined as well. A domain in Rigi specifies what entities in its input set are to be represented as nodes and which ones as arcs. The domain defined here closely resembles the OMT constructs such as classes, references, inheritance and so forth as well as the predefined C++ domain distributed with Rigi. In fact the domain used for the visualizations presented in this thesis is based on a subset of the C++ domain.

Integration of other visualization and reverse engineering tools is achieved by devising formatting clauses that either generate complete or partial input to the tool by appropriately formatting the information present in the repository.

4.3 Discussion

One of the limitations of the system described above is a performance bottleneck within the Prolog runtime environment. The runtime of the individual pattern queries appear to increase super linear with respect the number of facts in the repository. Although various optimization techniques have been applied to improve performance, scalability of the system is limited mainly by the pattern searching mechanism. However, Prolog provides powerful and easy to use tools to model complex relationships and is hence a superb tool for prototyping search engines. Once a search engine has been devised and the data model is fixed, the option remains to develop a tailor-made search engine using an industrial strength programming language such as C++. Although some of the flexibility to extend the data model and make use of the full power of logic programming will be lost, the benefits outweigh the losses.

Another notable restriction is that the entire system is based on only reverse engineering the structural information from the source code. Patterns displaying runtime behaviors, such as most behavioral patterns, can currently only be represented and detected by their structural aspect. A modification of the data model is required to include various runtime behaviors, raising two important issues:

- Runtime behavior is extremely difficult to extract and can not be done by a source code parser. It requires monitoring the system while it is running so that data about call sequences, execution time and so forth can be collected. It has to be assured that each line of source code is executed, so that a complete set of data is collected.
- Integration of the data collected by a potential runtime monitor with the current data model may prove to be very difficult as well. Call sequencing charts can be extremely complex and the resulting data volume may result in an unacceptable runtime performance of the already limited Prolog search engine.

Modifying the introduced system to incorporate runtime behavior is non-trivial. Currently only the mechanisms required to specify the structural aspects of a design pattern are provided.

Attempting to use off-the-shelf components often presents a fair number of integration problems. Many of the hurdles encountered here occurred in conjunction with the source code parser. Besides minor problems with compilation and installation, the parser turned out to be not as robust as one may expect from an industrial source code parser. Various aspects of the C++ language such as templates were not supported and it had difficulties in correctly parsing less frequently occurring language constructs.

Rigi as well as the Prolog runtime environment were easily integrated.

5 Fact Extraction

Reverse engineering is the process of extracting design information out of existing software systems [12]. Although various reverse engineering strategies have been proposed and implemented so far [22], and there exists a well defined methodology of how to reverse engineer, no precise definition of what to reverse engineer exists, or even can exist. The software artifacts extracted are first of all domain language specific and secondarily, dependent on the reverse engineering task at hand.

In our approach object oriented design patterns are specified using OMT and it is these constructs that are extracted from the source code. The source code is effectively reverse engineered into a large OMT diagram, represented as a collection of Prolog language facts. Theoretically, given the proper layout algorithm, such as the ones found in VLSI layout tools, one could use the reverse engineered information to represent the entire program as a large OMT diagram.

5.1 The Source Coder Parser

Unfortunately not all of the C++ language constructs are supported by *cppp*, causing it to either crash, terminate prematurely or produce structurally faulty ASTs whenever such language constructs are encountered. Source code examples therefore avoid the use of unsupported C++ constructs. In the cases where third party source code was parsed, most of the unsupported features could be replaced by other language constructs, which

generally changed the syntax but preserved the semantic properties of the program.

In order to keep the volume of data extracted to a minimum, a decision about what structural information is relevant had to be made. The following is a listing of all the language artifacts that were ignored during the parsing process.

- Global functions and variables that are not members of classes.
- Member variables that are not classes or references to such.
- Built-in operators and conversions.
- Implicit object construction and deletion.
- Type conversions.
- Destructors.

Although many of these artifacts represent OMT constructs such as functions and references, they are special cases. None of the object oriented design patterns discussed here make reference to any of these constructs. They can therefore be ignored.

5.2 The Fact Generator

The Fact Generator produces domain knowledge about the system by traversing its AST and extracting OMT constructs in the form of Prolog language facts. The facts generated are named according to the OMT construct they represent:

- Class
- Inheritance
- Call
- Operation
- Association
- Aggregation
- Instantiation

Each fact encapsulates enough information to uniquely identify the OMT construct it represents. That information is stored by the individual atoms within each fact. It encapsulates the names of abstractions in the source code that are involved in a particular

OMT construct, such as class names, function names, types and so forth.

During the assembly of a Prolog language fact, relationships that are not explicitly named within the source code are encountered and need to be expressed. The notion of an abstract class, for example, is captured within a `class` fact. However, the label *abstract* is not explicitly named within the source code and the property is derived from the number of virtual functions present in a class. A set of key words as shown in Table 5.1 has been defined to express this property as well as various others which are not explicitly named.

In addition to key words, character sequences used as prefixes to class and function names are required as well. Often a class name starts with a capital letter and can therefore not directly be adapted by a Prolog language statement, since the name will be interpreted as a variable. The character sequences *cl_* and *fn_* are used as prefixes for class and function names, ensuring that the names always start with a lower case letter.

Identifier	Meaning
cl_	preceeds class names
fn_	preceeds function names
concrete	concrete class or non-virtual function
virtual	virtual function
abstract	abstract class
ref	pass by reference parameter passing
val	pass by value parameter passing
int, float, double ...	simple types
one, many	reference cardinality

Table 5.1 Reserved words

The text file containing the Prolog facts is eventually read into the Prolog runtime environment, where queries in the form of Prolog clauses then attempt to identify the facts necessary to form an occurrence of a pattern.

Following is a brief description of all the program artifacts that are extracted from the AST and the corresponding Prolog language facts that are added to the repository. The facts created stand in an almost one-to-one correspondence with the OMT constructs introduced in the previous chapter.

5.2.1 Classes and Inheritance

The classes in a program are represented by two types of facts: `class` and `inheritance`. `Class` facts are composed of names and a declaration specifier of either `virtual` or `concrete` and `inheritance` facts are composed of base and super class names.

Example source code:

```
class hoo { };
class foo : public hoo {
public :
        class bar {};
};
```

Parser output :

```
class(cl_bar,concrete).
class(cl_foo,concrete).
class(cl_hoo,concrete).
inheritance(cl_foo,cl_hoo)
```

The example shows a set of three classes that include nesting and inheritance relations. The parser outputs three `class` and one `inheritance` facts. The declaration specifiers of all classes are correctly determined to be concrete. Note that none of the patterns analyzed here concerns itself with class scopes. The nested class `bar` therefore produces the same fact as the other classes.

5.2.2 Association and Aggregation

Object references are maintained by member variables, the types of which are either pointers or references or entire objects. The two types of references are referred to as association and aggregation, respectively. A reference is always associated with a cardinality of either `one` or `many`, which indicates how many objects are referenced. Array structures always have a cardinality of `many` and pointers and C++ references always have a cardinality of `one`. In the cases where a pointer is used instead of an array to keep a reference to a collection of objects, the Fact Generator unfortunately only detects a cardinality of one.

Example source code:

```
class foo {};
class bar {

        foo *f1;                (1)
        foo f2;                 (2)
        foo &f3;                (3)
        foo *a1[12];            (4)
        foo a2[12];             (5)
};
```

Parser output:

```
class(cl_bar,concrete).
class(cl_foo,concrete).
aggregation(cl_bar,cl_foo,many).     (5)
association(cl_bar,cl_foo,many).     (4)
association(cl_bar,cl_foo,one).      (3)
aggregation(cl_bar,cl_foo,one).      (2)
association(cl_bar,cl_foo,one).      (1)
```

A reference fact is either named `association` or `aggregation`. Such a fact contains three Prolog atoms, the names of the referencing class, referenced class and the cardinality. Shown above are the facts generated by parsing a simple class declaration. The numbers given at the end of the lines are used to correlate the facts with the corresponding source code artifacts.

5.2.3 Operation

Class member functions (methods) are the only type of function extracted by the parser. An operation fact is made up of a class name (of which the function is a member), a function name, a declaration specifier of either `virtual` or `concrete`, a list of parameter types and a list of return value types. The manner in which the function parameters are represented sets the operation fact apart from any other type of fact.

Any number of parameters of varying types may be passed to a function. Representation of a variable amount of information within a single fact requires the use of the Prolog language *list* construct. A list consists of a set of tuples separated by commas. Each function parameter is represented by a tuple of `(declaration specifier, type)`. The `declaration specifier`, having a value of either `ptr` or `val`, indicates *pass-by-value* or *pass-by-reference*. The `type` names the parameter type, which is either a class name or a simple type name. In the case of an ellipses parameter [23] a function can have any number of parameters, all specified at runtime. The parameter list only contains a single tuple of variables `(X,Y)`, effectively creating a wild card parameter list.

Although there can only be one return value per function, they are also represented as a list of tuples. However, only one tuple will be in the list at all times. The motivation behind this representation is that the same list traversal clauses can be used to extract parameter and return value information from a fact.

Example source code:

```
class foo {

        foo() {};
        ~foo() {};

        foo & operator=(foo &src) { return *this;};
        foo & operator[](int i) { return *this;};

        void f(foo *F, ...)  {};

};
```

Parser OutPut:

```
class(cl_foo,concrete).
operation(cl_foo,fn_f,concrete,[(ptr,cl_foo),(X,Y)],[(val,void)]).
operation(cl_foo,fn_operatorSq,concrete,[(val,int)],[(ref,cl_foo)]).
operation(cl_foo,fn_operatorEq,concrete,[(ref,cl_foo)],[(ref,cl_foo)]).
operation(cl_foo,fn_foo,concrete,[],[(val,int)]).
```

The example above shows the facts generated by a class with five member functions. It illustrates various types of class member functions. The names of overloaded operators such as `operator()` and `operator[]` are changed to `operaterEq` and `operatorSq`. Prolog only allows for alphanumeric characters to appear in the name of a clause, variable and atom. The operators `()` and `[]` are used to describe lists and sets. Improper usage of these characters causes the Prolog interpreter to report errors. All overloaded operator names are changed in this manner.

The constructor and destructor of a class produce no operation facts. Constructors and destructors are member functions that are always present, either explicitly as shown above or implicitly, that is created by the compiler. However, the function bodies of constructors and destructors are still parsed.

5.2.4 Function Calls

A function call may be nested within a variety of programming language constructs. The following example illustrates how even deeply nested function calls are extracted. A fact generated by a function call contains information about names and member classes of caller and callee. Global function calls are filtered out by the parser and therefore produce no facts.

Example Source Code

```
class foo {
public :
        foo * f(foo *);
        int h(foo *);
};
foo * foo::f(foo * aFoo) {
```

```
        return aFoo;
};
void foo::h(foo * aFoo) {
        return 1;
};

class bar {
public :

        void g(foo *);
};

void bar::g(foo *f) {
foo *aFoo;
while(aFoo->h(aFoo->f(aFoo)))
                g(aFoo);
};
```

Parser Output

```
class(cl_bar,concrete).
class(cl_foo,concrete).
operation(cl_bar,fn_g,concrete,[(ptr,cl_foo)],[(val,void)]).
operation(cl_foo,fn_h,concrete,[(ptr,cl_foo)],[(val,int)]).
operation(cl_foo,fn_f,concrete,[(ptr,cl_foo)],[(ptr,cl_foo)]).
call(cl_bar,fn_g,cl_bar,fn_g).
call(cl_bar,fn_g,cl_foo,fn_f).
call(cl_bar,fn_g,cl_foo,fn_h).
```

The example shows a nested sequence of function calls within the while statement of function `bar::g`. Other than the `class` and `operation` facts, three `call` facts are produced, identifying the classes to which the called functions belong.

5.2.5 Instantiations

An instantiation occurs when an instance of a class (i.e., an object) is created. There are two types of instantiations: implicit and explicit. Implicit instantiation occurs during pass by value of objects, creation of temporary objects or during instantiation of base classes and aggregated objects. Explicit instantiation is controlled by the programmer and mostly occurs in conjunction with the `operator` With respect to design patterns only these instantiations are of importance. All other object instantiations therefore generate no facts.

```
class foo {};
class bar {

foo *aFoo1;
foo aFoo2;

public:
        bar();
};

bar::bar() : aFoo2() {

        aFoo1 = new foo();
};
class(cl_bar,concrete).
class(cl_foo,concrete).
operation(cl_bar,fn_bar,concrete,[],[(val,int)]).
instantiation(cl_bar,cl_foo).
aggregation(cl_bar,cl_foo,one).
association(cl_bar,cl_foo,one).
```

An `instantiation` fact contains the names of the instantiating and instantiated classes.

5.3 Summary

The fact extraction system, implemented by the parser and Fact Generator effectively decomposes the entire source code of a C++ program by extracting every occurrence of six different types of OMT constructs. These constructs are then stored in the data repository, where they serve as the domain of the subject system. Depending on the construct extracted, information to identify the construct uniquely is extracted and stored as well.

A decision about whether to consider certain language constructs as candidates for extraction had to be made in several cases. All occurrences of implicit, compiler generated relationships such as, implicit object creation and destruction and conversion operator invocations are ignored during fact extraction. Although these software artifacts represent OMT constructs, their occurrences are seldom described in OMT representations of a

design pattern. For reasons of simplicity and runtime performance these constructs are ignored.

Any invocation of global functions is ignored as well. Global functions are not enclosed by a class scope and no class name exists that could be inserted into the `operation` and `call` facts pertaining to a global function. Although the global scope could be considered its own class and be named accordingly, a decision against that alternative was made. This may eventually result in false positive identification of a design pattern that contains that manufactured global class as one of its participants.

6 Pattern Identification

This chapter presents the query mechanisms used to recover design patterns from the data repository. The previous chapter described how the fact extraction system uncovered a set of OMT constructs from the source code in the form of Prolog language facts. The task that remains now is to specify a particular arrangement of OMT constructs, also referred to as an object model, and determine whether a subset of the constructs representing the domain knowledge matches the arrangement. The object models, which in this case are design patterns, are specified as conjunctive Prolog clauses, each stating the composition of a pattern in terms of the Prolog language facts defined in the data model.

By executing a clause in the runtime environment an attempt is made to match the arrangement specified by the clause with the facts present in the data repository. Once a match has been found, the presence of the object model within the source code is confirmed. The process is similar to visually finding a certain arrangement of OMT constructs in a diagram. This is not always an easy task, especially for very large diagrams

6.1 Overview

This section presents the process of designing a query from a simple object model. Relevant OMT constructs in the diagrammatic representation are identified and then converted into a Prolog clause.

Figure 6.1 shows the *department store* object model. It is specified using the OMT

constructs introduced in Chapter 2. The model consists of three classes named *Customer*, *Employee* and a *Supervisor*. The *Employee* references a *Customer* who is serviced by calling the *Employee's* member function `Service(Customer& C)`. A *Supervisor*, who is also an *Employee*, (implemented by the inheritance of *Employee* by the *Supervisor*) maintains a reference to another, subordinate *Employee*. Whenever the *Supervisor* services a *Customer* by calling its own `Service(Customer& C)` function the work is delegated by calling the `Service(Customer& C)` function of the referenced *Employee*.

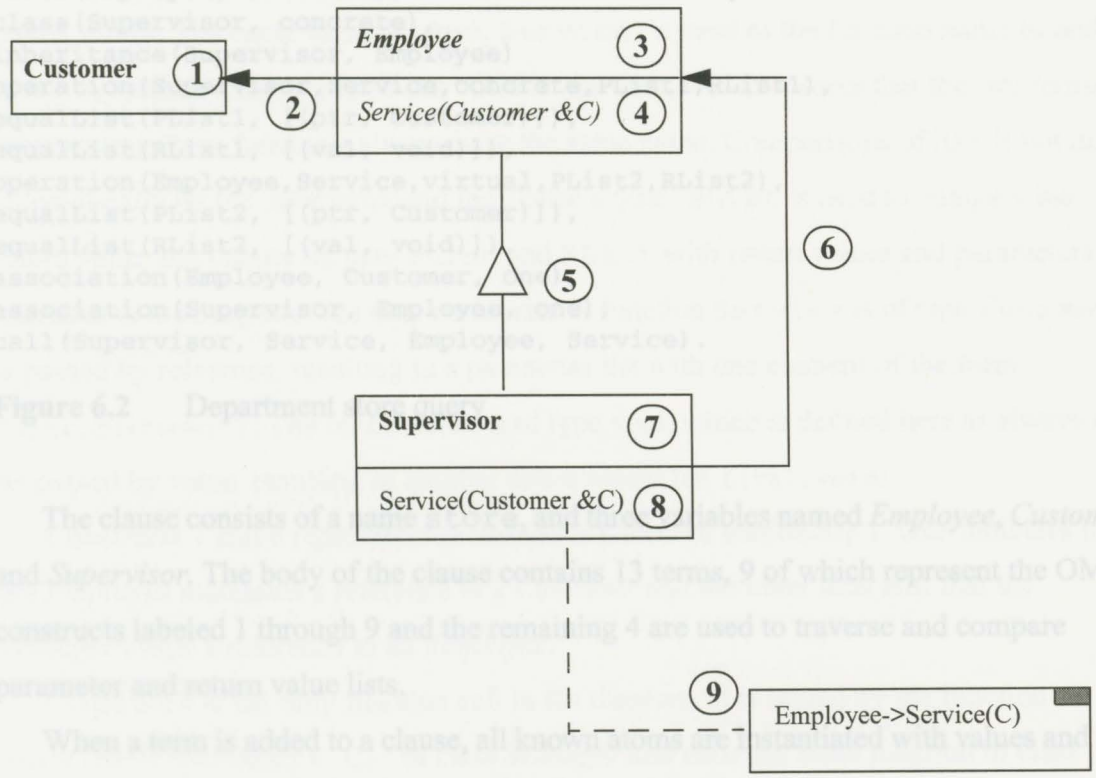

Figure 6.1 Department store object model

Setting aside issues about the effectiveness and usefulness of the above object model,

attention is now turned to translating the OMT diagram into a Prolog clause. The clause consists of a head and a body. The head is composed of a clause name and a set of variables, each of which corresponds to a class in the object model. The body of the clause is composed of conjunctive arrangements of terms, each of which is a Prolog language fact representing an OMT construct. Figure 6.2 displays the Prolog clause derived from Figure 6.1. The number at the end of a line in Figure 6.2 corresponds to the number given to the corresponding OMT constructs in Figure 6.1.

```
store (Customer, Employee, Supervisor) :-

class(Customer,concrete),                                          #1
class(Employee, abstract),                                         #3
class(Supervisor, concrete),                                       #7
inheritance(Supervisor, Employee)                                  #5
operation(Supervisor,Service,concrete,PList1,RList1),              #4
equalList(PList1, [(ptr, Customer)]),
equalList(RList1, [(val, void)]),
operation(Employee,Service,virtual,PList2,RList2),                 #8
equalList(PList2, [(ptr, Customer)]),
equalList(RList2, [(val, void)]),
association(Employee, Customer, one),                              #2
association(Supervisor, Employee, one),                            #6
call(Supervisor, Service, Employee, Service).                      #9
```

Figure 6.2 Department store query

The clause consists of a name `store`, and three variables named *Employee*, *Customer* and *Supervisor*. The body of the clause contains 13 terms, 9 of which represent the OMT constructs labeled 1 through 9 and the remaining 4 are used to traverse and compare parameter and return value lists.

When a term is added to a clause, all known atoms are instantiated with values and the atoms that are not known are substituted with variables. Class names and function names, for example, are never instantiated, as these are inherent to the source code and assigned values during the search performed by the Prolog runtime environment. Declaration specifiers of functions, are always known and substituted with either `concrete` or `virtual`. During the instantiation process the Prolog interpreter immediately dismisses all those facts that do not match in all instantiated atoms.

OMT constructs numbered 1, 3 and 7 are classes. The atoms in a `class` fact represent the class name and the declaration specifier of either `abstract` or `concrete`, which is indicated in the OMT diagram by either a class name written in plain or italicized characters. The class names are represented by variables *Customer*, *Employee* and *Manager*. The declaration specifiers are instantiated as either `concrete` or `abstract` depending on the class.

Construct 5 is the only inheritance present in the diagram. Class *Supervisor* is a subclass of class *Employee*, thus one inheritance term is added to the clause..

Constructs 4 and 8 represent two functions. One is a member of class *Supervisor* and the other is a member of class *Employee*. Both have identical names, parameters and return values. The same variable name, `Service`, is used as the function name in both terms. A variable can only have one value at the time, which ensures that the two terms representing these functions always have the same name. Comparisons of lists is not done automatically by the interpreter and the clause `equalsList` is used to compare the return value and parameter lists `PList` and `RList` with return values and parameters specified in the diagram. The only parameter to function `Service` is of type *Customer*. It is passed by reference, resulting in a parameter list with one element of the form `[(ptr,Customer)]`. The return value is of type void, which is defined here as always to be passed by value, resulting in another one-element list `[(val,void)]`.

Constructs 7 and 6 represent two association facts of cardinality 1. One indicates that the *Employee* maintains a reference to a *Customer* and the other indicates that the *Manager* keeps a reference to an *Employee*.

Construct 9 is the only function call in the diagram. It is issued by the function `Service(Customer& C)` of class *Manager* and calls the same function of class *Employee*. Here it has been interpreted as a call, however, it may as well be a delegation.

All relevant OMT constructs in the diagram have been identified and added to the `store` clause, which now effectively models the diagram. The clause can be used to identify the *department store* object model within any piece of C++ source code, given that the extracted facts have been read into the Sicstus runtime environment.

A clause is satisfied when all variables in its body and head are instantiated. The

instantiation process starts by picking the first term in the clause. All the facts in the repository that are of the same type as the term are then considered. The ones that do not match the already instantiated atoms are further eliminated. For example, a fact `class(bar, concrete)` is not considered when instantiating variable `AClass` in `class(AClass, virtual)`, as the facts differ in the atom at the second position. The process continues by instantiating all variables in the first term with the values found in a potentially matching fact, effectively binding the variable to values. Next, the second term is considered in a similar manner. The process continues until all variables are successfully instantiated, or until all facts have been considered and no instantiation has been found.

6.2 A Clause Toolbox

This section introduces a set of clauses that will prove helpful when modeling object - oriented design patterns. Object oriented design patterns are a special type of object model and the corresponding OMT representations are often too restrictive to capture possible variations of the pattern.

6.2.1 Distant Ancestry Relationships

Only inheritance relationships which capture the occurrence of one class inheriting another are extracted from the source code. However, sometimes it is of interest whether or not two classes are related by an inheritance over several layers of classes.

```
subclass(ClassA,ClassX):-
        inheritance(ClassA,ClassX);
        (
                inheritance(ClassA,ClassY),
                subclass(ClassY,ClassX)
        ).

superclass(ClassA,ClassX):-
        inheritance(ClassX,ClassA);
        (
                inheritance(ClassX,ClassY),
                superclass(ClassA,ClassY)
        ).
```

Figure 6.3 Ancestry relationship clause

Figure 6.3 presents two clauses which model the mentioned ancestry relationships. The clauses `subclass` or `superclass` attempt to recursively satisfy the inheritance facts. The `subclass(ClassA, ClassX)`, for example, is satisified if either `ClassA` directly inherits `ClassX`, or if there exists a `ClassY` such that `ClassA` inherits `ClassY` and `ClassY` is in turn a subclass of `ClassX`.

6.2.2 Call Delegation

Call delegation was discussed in section 2.2.4. It refers to one function delegating its invocation by calling another function. This mechanism is often used in object composition. Call delegation is not directly extracted from the source code, but the relationship can be described and detected by the clause shown in Figure 6.4. With respect to design patterns it is often the case that function calls are actually intended to be call delegations. However, there are no OMT constructs that explicitly distinguish calls and delegations, and the property is therefore not explicitly present in the pattern's OMT diagram.

```
delegation(C1,F1,C2,F2):-
        call(C1,F1,C2,F2),
        setof((X,Y),call(C1,F1,X,Y),S),
        size(S,N),
        +N =:= 1.
```

Figure 6.4 Call delegation clause

The clause is satisfied if the method F1 of class C1 calls method F2 of class C2, and the size of the set of all the functions invoked by function F1 is equal to 1. In other words the clause is only then satisfied when function F1 issues a single function call. Note that a single function invocation within a control structure such as a loop will also be interpreted as a delegation.

6.2.3 Indirect Association

An indirect association is present when one object references another via some kind of container class. Consider the example where a class A references a Stack of objects of type B. Conceptually many objects of type B are referenced. In reality however, only one object of type Stack is referenced. Class A then accesses objects of type B via access functions provided by the Stack. Many patterns that contain references of cardinality larger than 1 may in fact be implemented via a container or manager class.

```
indirectReference(A,B):-
        association(A,X,_)
        call(A,SomeOp,X,Operation),
        operation(X,Operation,_,PList,RList),
        (
                (
                        find((ptr,B),PList);
                        find((_,B),RList)
                );
                (
                        subclass(B,Y),
                        (
                        find((ptr,Y),PList);
                        find((_,Y),RList)
                        )
                )
        ).
```

Figure 6.5 Indirect association clause

The rule defines an indirect reference from class A to class B present if

- Class A references a class X, which in turn references B.
- Class X provides some operation that either takes as an argument or returns a refer-

ence to Class B or a base class of Class B.

The first condition simply determines whether the references among the classes are sufficient to even assume an indirect reference. That is only the case when A references X and X references B. The second condition states that every container or manager class must also provide access functions, where elements are exchanged by either reference or value. Moreover, intrusive container classes may only handle base types defined by the container class. Therefore the access functions may only operate on base types of B.

6.3 Pattern Clauses

The following are Prolog clauses used to identify occurrences of the patterns introduced in Chapter 2. The rules were designed by modeling the OMT representations of each pattern.

In some cases the diagrams are not modeled exactly and some specifications are relaxed. This is especially true for the classes that are specified to be concrete. Experiments have shown that a pattern is often present except that one of the classes specified as concrete contains virtual functions. This may, for example, be the case when one class participates in two different patterns. The pattern is still present but it will not be interpreted as such by the search. The classes are therefore allowed to be either concrete or abstract. There is much room for variation when specifying design patterns, as they are merely design guidelines.

Several of the terms in a clause, especially the occurrences where the specifications have been relaxed, contain free variables indicated by an underscore (_). These are indications that a particular aspect of a term is not relevant with respect to the pattern. Often, for example, the parameters and return values of an operation are of no interest; only the existence of the operation is of importance. The Prolog runtime environment substitutes the free variables with any value required to satisfy a clause, but never instantiates the underscore with that value.

Additional terms are added to prevent the assignment of the same class to two different

variables in the clause. A single class can not occur in the same pattern multiple times. However, the search engine does not restrict classes from being participants in several different design patterns.

The order in which the terms in a clause are arranged, maximizes the runtime performance of the Prolog interpreter by keeping the width of the substitution tree minimal [10]. Experiments have shown that the runtime performance of the searches fluctuates depending on the order in which the terms are arranged.

6.3.1 Adapter

The adapter clause shown in Figure 6.6 closely models Figure 2.5. The clause has four variables, each of which represents a class in the pattern.

```
adapter(Client,Target,Adapter,Adaptee):-
        inheritance(Adapter,Target),
        association(Adapter,Adaptee,one),
        association(Client,Target,_),
        class(Target,abstract),
        class(Adaptee,_),
        class(Adapter,_),
        class(Client,_),
        delegation(Adapter,Request,Adaptee,SpecificRequest),
        operation(Target,Request,virtual,_,_),
        operation(Adapter,Request,concrete,_,_),
        operation(Adaptee,SpecificRequest,concrete,_,_),
        Target \== Adapter,
        Target \== Adaptee,
        Adapter \== Adaptee.
```

Figure 6.6 Adapter clause

Although the diagram specifies that the *Client*, *Adapter* and *Adaptee* classes are strictly concrete, that constraint has been relaxed here and the corresponding atoms in the `class` facts has been replaced by a free variable. The cardinality of the association from *Client* to *Target* has been relaxed as well. It is of no importance to the pattern whether the *Client* is abstract or not, or whether it references one or many Targets. One could also argue that the *Client* class can be completely ignored.

Instead of modeling the function call from function *Request* to *SpecificRequest* using a `call` fact, the `delegation` clause has been used. It will assure that the call is delegated as intended by the pattern.

6.3.2 Bridge

The Bridge Prolog clause displayed in Figure 6.7 contains only four variables and not five as Figure 2.6 suggests. The OMT diagram shows a *ConcreteImplementorA* and a *ConcreteImplementorB* class. The intent is to show that there may be several *ConcreteImplementors*. Within the Prolog clause, however, one variable will be sufficient to find all *ConcreteImplementors* present in an instance of the pattern. There will be one tuple for each *ConcreteImplementor* found.

```
abridge(Abstraction,RefinedAbstraction,Implementor,ConcreteImpl):-
        association(Abstraction,Implementor,one),
        inheritance(RefinedAbstraction,Abstraction),
        inheritance(ConcreteImpl,Implementor),
        class(Abstraction,abstract),
        class(RefinedAbstraction,_),
        class(Implementor,abstract),
        class(ConcreteImpl,_),
        Abstraction \== Implementor,
        Abstraction \== RefinedAbstraction,
        Abstraction \== ConcreteImp,
        RefinedAbstraction \== ConcreteImpl,
        delegation(Abstraction,Operation,Implementor,OperationImp),
        operation(Abstraction,Operation,concrete,_,_),
        operation(Implementor,OperationImp,virtual,_,_),
        operation(ConcreteImpl,OperationImp,concrete,_,_).
```

Figure 6.7 Bridge clause

Furthermore, the clause allows the *RefinedAbstraction* and *ConcreteImplementor* to be either concrete or abstract. *RefinedAbstraction* and/or *ConcreteImplementors* may contain virtual functions making the class abstract. It is quite possible for a *ConreteImplementor* to specify an interface for a whole family of *ConcreteImplementors*. Neglecting to relax these conditions will cause instances of the pattern to be missed.

The call `delegation` clause replaces the `call` fact to indicate the function call

from function *Operation* to function *OperationImp*. Again, the intent of this call is to delegate.

6.3.3 Composite

Due to runtime considerations the Composite pattern clause has been divided into three separate ones. Each clause specifies the conditions that the classes *Component*, *Composite* or *Leaf* have to meet in order to be part of the pattern. Note that these clauses are embedded in a Prolog program that performs further processing. Each of the clauses below produces a set of tuples which then have to be combined. Many classes satisfy each clause individually, but only a few satisfy all three.

```
component(Component,Operation,Add,Remove,GetChild):-
        class(Component,_),
        operation(Component,Operation,virtual,_,_),
        operation(Component,Add,virtual,AddPList,_),
        operation(Component,Remove,virtual,RemovePList,_),
        operation(Component,GetChild,virtual,GetChildPList,
                                                GetChildRList),
        Add \== Remove,
        Operation \== Add,
        Operation \== Remove,
        Operation \== GetChild,
        find((_,Component),AddPList),
        find((_,Component),RemovePList),
        find((val,int),GetChildPList),
        find((_,Component),GetChildRList).

composite(Component,Composite):-
        componentTuple(Component,Operation,Add,Remove,GetChild),
        inheritance(Composite,Component),
        call(Composite,Operation,Component,Operation),
        class(Composite,_),
        operation(Composite,Operation,concrete,_,_),
        operation(Composite,Add,concrete,AddPList,_),
        operation(Composite,Remove,concrete,RemovePList,_),
        operation(Composite,GetChild,concrete,GetChildPList,
                                                GetChildRList),
        find((_,Component),AddPList),
        find((_,Component),RemovePList),
        find((val,int),GetChildPList),
```

```
        find((_,Component),GetChildRList),
        (
                association(Composite,Component,many);
                indirectReference(Composite,Component)
        ).

leaf(Component,Leaf):-
        componentTuple(Component,Operation,_,_,_),
        inheritance(Leaf,Component),
        class(Leaf,_),
        operation(Leaf,Operation,concrete,_,_),
        not(composite(_,Leaf)).
```

Figure 6.8 Composite clauses

The first term in the clauses `leaf` and `composite` makes reference to a type of fact that has not yet been introduced. The following fact is added to the repository for each successful tuple that satisfies the `component` clause, effectively capturing the result of the clause.

```
componentTuple(Component, Operation, Add, Remove, GetChild)
```

The `component` clause is executed first and all `componentTuple` facts are added before the clauses `leaf` and `component` are executed.

The rules specifying the *Leaf* and *Composite* classes as strictly concrete have been relaxed. The association from the *Composite* to the *Component* has been replaced by an indirect association as introduced in section 6.2.3. The latter is motivated by the fact that it is a standard programming technique to encapsulate the maintenance of references to many objects using a container class of some sort.

The parameter and return values to the functions in this pattern are of importance as well. The clause `find` is used to determine whether a list contains a specific element or not.

6.3.4 Command

Similar to the Adapter and Bridge, the Command clause has been directly modeled from

the OMT representation of the pattern.

```
aCommand(Client,Receiver,ConcreteCommand,Command,Invoker):-
        inheritance(ConcreteCommand,Command),
        association(Client,Receiver,_),
        class(Client,_),
        class(Receiver,_),
        association(Invoker,Command,_),
        class(Invoker,_),
        class(Command,abstract),
        association(ConcreteCommand,Receiver,one),
        class(ConcreteCommand,_),
        ConcreteCommand \== Invoker,
        ConcreteCommand \== Receiver,
        ConcreteCommand \== Client,
        Receiver \== Invoker,
        Receiver \== Client,
        Client \== Invoker,
        Client \== Command,
        instantiation(Client,ConcreteCommand),
        delegation(ConcreteCommand,Execute,Receiver,Action),
        operation(ConcreteCommand,Execute,concrete,_,_),
        operation(Command,Execute,virtual,_,_),
        operation(Receiver,Action,concrete,_,_).
```

Figure 6.9 Command clause

The requirements for the classes *Client*, *Invoker*, *Receiver* and *ConcreteCommand* to be strictly concrete has again been relaxed for the same reasons as described earlier. No reason could be found for any of these classes to be concrete as opposed to virtual.

A call `delegation` clause has been used to represent the call from function `Execute` to function `Action`.

6.4 Summary

This chapter described how an OMT diagram is translated into a Prolog clause. Such a clause can subsequently be used to detect the presence of the object model presented in the diagram among the facts contained within the data repository. The process of composing a clause includes identifying all relevant OMT constructs in a diagram and conjunctively adding each to the clause.

Selected patterns obtained by Gamma *et al.* [2] which were discussed in Chapter 2 are translated into Prolog clauses using their object model diagrams. However, many of the restrictions imposed by some of the object models are relaxed. Since OMT class constructs always specify a class as abstract or virtual it follows that all classes involved in a pattern are strictly one or the other. However, this restriction defeats the purpose, as it is often irrelevant whether a class is concrete or abstract, just as it is often irrelevant what the return value of a function is.

7 Pattern Processing

This chapter explains how the Prolog clauses introduced previously are applied to uncover object oriented design patterns. A set of the class tuples that is retrieved by a clause is further processed to produce a unique instance of a pattern. As not every tuple necessarily results in a pattern, a two step *consolidation* process is applied to combine the tuples. The first stage adds a new fact to the repository for every tuple found. In the second stage the tuples, now represented as facts, are combined according to a rule in the form of a regular expression and a set of clauses is specially designed to model the regular expression.

The processing of patterns not only includes the identification but also the presentation of pattern instances. The software visualization tool Rigi has been chosen to present the extracted design patterns. Usage of the tool requires that input in the proper format is generated. Several Prolog clauses are used to generate input describing domain and pattern information.

Throughout this chapter various example clauses pertaining to the Bridge pattern are introduced. However, the same set of clauses also exists for the remaining patterns. Please refer to Appendix A, B and C for a complete listing of all Prolog clauses used during the pattern processing step.

7.1 Primary Processing

In this section an overview is provided of the steps necessary before a pattern clause can

be executed. First, the source code parser and fact generator reverse engineer the subject system. The resulting *fact database* and the *clause database* are then imported into the Prolog runtime environment. All necessary information is now present and clauses are ready to execute.

When a clause is executed by the Prolog runtime environment it produces tuples of variable instantiations satisfying the clause one at a time. In order to capture a complete set of tuples the Prolog language `setof` construct is applied. Once the set of tuples is constructed the `assert` statement is used to add each tuple as a new fact to the repository. Figure 7.1 illustrates the clauses necessary to identify the bridge tuples and add each as a new fact to the repository.

```
produceTuples:-
        setof((X,Y,Z,W),bridge(X,Y,Z,W),RESULT),
        addTuples(RESULT).

addTuples((X,Y,Z,W) | Xs) :-
        assert(bridgeTuple(X,Y,Z,W)),

addTuples([]).
```

Figure 7.1 Bridge pattern primary processing clauses

The Prolog clause `produceTuples` first executes the `setof` construct and then passes the set `RESULT` to the `addTuples` clause. The set contains all tuples of classes that satisfy the bridge pattern clauses introduced in section 6.3.2. The `addTuples` clause then recursively traverses `RESULT` and executes the `assert` statement for each tuple. A new fact called `bridgeTuple` with the classes W, X, Y and Z as its atoms is added to the repository.

A clause like the one in Figure 7.1 exists for every pattern considered. Appendix A presents a complete listing of the primary processing clauses. Once all the preprocessing steps have been performed for each pattern, a new basis for subsequent processing has been built.

7.2 Secondary Processing

After the primary processing has been performed it remains to be determined which tuples actually belong to the same instance of a pattern. First, a set of rules has to be devised that define exactly how to combine individual tuples to instances, and second, these rules have to be translated into Prolog clauses. This section discusses the reasons why these rules are necessary and how they are implemented.

7.2.1 Pattern Consolidation Rules

Pattern consolidation rules is the term given to a set of regular expressions that guide the process of extracting pattern instances. Often a large number of tuples are produced for each instance of a pattern, which is due to the fact that several classes with identical properties can be involved in a pattern. Although these classes are specified only once in the OMT diagram, the intent is that there can be many.

In the case of the Bridge pattern, for example, the *ConcreteImplementor* is a an abstract definition of an entire set of classes, each of which displays the properties of a *ConcreteImplementor*. The primary processing step generates the following fact (tuple) for every *ConreteImplementor* it finds.

```
bridgeTuple (Abstraction, RefinedAbstraction,
                Implementor, ConcreteImplementor).
```

However, in many of these facts only the value of the *ConcreteImplementor* differs. It is clear that none of the tuples differing in only the *ConcreteImplementor* compose a unique Bridge pattern. The same observation can be made with respect to the *RefinedAbstraction*. It represents a set of classes, all of which share the property to have the *Abstraction* as a base class.

A remaining problem is to devise the rule that dictates which of the tuples belongs to the same instance, and which ones belong to another instance. Consider the sets of classes in Table 7.1. Assume that every row in the table represents a `bridgeTuple` fact

produced during primary processing.

Abstraction	Refined Abstraction	Implementor	Concrete Implementor
Port	PrintPort	Ink	Bitmap
Port	PrintPort	Ink	RGBColor
Port	WindowPort	Ink	Bitmap
Port	WindowPort	Ink	RGBColor

Table 7.1 Example Bridge 1

The *Abstractions* and *Implementors* are the same in all four of the tuples and only the *RefinedAbstractions* and *ConcreteImplementors* vary. The Bridge pattern, however, explicitly states that it is possible to have more than one *ConcreteImplementor* (*ConcreteImplementorA* and *ConcreteImplementorB* in Figure 2.6) as well as several *RefinedAbstraction* per pattern. The above tuples therefore compose only a single instance of the Bridge pattern, which is made up of

1 *Abstraction*,

2 *RefinedAbstractions*,

1 *Implementor*,

2 *ConcreteImplementors*.

The previous example shows how several tuples are combined to form a single instance of a pattern. The criteria applied in combining the tuples is that many *RefinedAbstractions* and *ConcreteImplementors* are allowed per instance. The criteria chosen seem sound. *Abstraction* and *RefinedAbstraction* as well as *Implementor* and *ConcreteImplementor* can be combined into a single class if a 1-to-1 relationship among them is intended. However, the tuple combining criteria are not always straightforward. Consider the set of tuples Table 7.2.

Abstraction	Refined Abstraction	Implementor	Concrete Implementor
StaticTextView	TextView	Text	CheapText
StaticTextView	TextView	Text	GapText
StaticTextView	TextView	TextPager	LinePager

Table 7.2 Example Bridge 2

In this example the tuples all share the same *Abstraction* and *RefinedAbstraction*. However, there are two different *Implementors* and consequently different *Concrete-Implementors*. In other words, the *Abstraction* provides an interface by combining two separate *Implementations*. It is almost like a three-way bridge built on the fork of two joining rivers. Are we dealing with one or two instances of the Bridge pattern? Certainly there will be divided opinions on that question. Here it has been decided to group these tuples into the same instance of the Bridge pattern, which is therefore composed of

1 *Abstraction*,

1 *RefinedAbstraction*,

2 *Implementors*,

2 *ConcreteImplementors*.

A regular expression in terms of the classes involved in a pattern provides an effective mechanism to specify the rules by which the class tuples are consolidated to pattern instances. The expression is implemented by a Prolog clause that produces a set of pattern instances. These pattern instances then provide the final product of the search.

Based on the observations presented above, a regular expression of the form *{{Abstraction, RefinedAbstraction*},{Implementor,ConcreteImplementor*}*}* is produced to define the tuple consolidation. In plain English, the rule can be expressed with the following sentences:

> Per Bridge pattern there may be 1 Abstraction, derived from which may be several RefinedAbstractions. The Abstraction may reference many Implementors. Derived

from each Implementor may be many ConcreteImplementors.

Tuples found for the Composite, Adapter and Command pattern are processed in a fashion similar to the one introduced for the Bridge pattern. Table 7.3 lists the regular expressions which guide the consolidation process for each type of the pattern.

Pattern	Regular Expression
Adapter	{{Adapter, Adaptee, Target} , Client*}
Bridge	{{Abstraction, RefinedAbstraction*},{Implementor,ConcreteImplementor*}*}
Composite	{Component,Composite*,Leaf*}
Command	{Command,{ConcreteCommand, {Receiver, Clients*} *,Invokers*}

Table 7.3 Pattern consolidation rules

Note that these rules are solely based on the author's interpretation of the patterns and can certainly be challenged.

7.2.2 Pattern Consolidation Clauses

Devising a Prolog clause that combines the `bridgeTuple` facts according to the above regular expression proved to be a non-trivial task. Figure 7.2 presents the set of Prolog clauses required to combine tuples to instances of the Bridge pattern.

```
processTuples(RESULT) :-
        setof(
                (A,R,I),
                (
                        implementors(A,I),
                        refinedAbstractions(A,R)
                ),
                RESULT
        ).
```

```
refinedAbstraction(A,R) :- bridgeTuple(A,R,_,_).
refinedAbstractions(A,RESULT) :-
        setof(
                R,
                refinedAbstraction(A,R),
                RESULT
        ).

implementor(A,I) :- bridgeTuple(A,_,I,_).
implementors(A,RESULT) :-
        setof((I,C),
        (
                implementor(A,I),
                concreteImplementors(A,I,C)
        ),
        RESULT
        ).

concreteImplementor(A,I,C) :- bridgeTuple(A,_,I,C).
concreteImplementors(A,I,RESULT) :-
        setof(
                C,
                concreteImplementor(A,I,C),
                RESULT
        ).
```

Figure 7.2 Bridge pattern secondary processing clauses

The `processTuples` clause produces the set `RESULT` in which each element is a multiset of classes in a format dictated by the regular expression. The set is the final product of the secondary processing step. It contains all instances of the Bridge pattern present in the source code of the subject system.

Prolog clauses performing the consolidation processes for the remaining patterns are presented in Appendix B. Each clause extracts the instances of a pattern by making use of the facts generated during primary processing.

7.3 Output Generation

This section primarily addresses the integration of the software visualization tool. The goal is to produce information such that the tool can effectively be used to display the object oriented design patterns among the original software artifacts that were extracted

from the source code. Rigi, the visualization tool used in this thesis, requires input in the form of Rigi Standard Format (RSF).

Using RSF, nodes and arcs are represented as tuples of information referred to as node tuples and arc tuples, respectively. Node tuples are always of a particular **nodetype** such as *Class* and *Operation*, and arc tuples are of an **arctype** such as *reference* or *inheritance*. A **nodetype** is followed by the name of the abstraction representing a node and an **arctype** is followed by the names of the abstractions that are connected by the arc. Figure 7.3 summarizes the two types of RSF tuples.

```
nodetype    name
arctype     source      destination
```

Figure 7.3 RSF tuple format

Two sets of Prolog clauses are needed to convert the fact database and the pattern instances to RSF. Converting the fact database requires that one RSF tuple be produced for each fact in the database. Figure 7.4 shows as an example the clauses required to convert the `class` facts to RSF node tuples. The clause `outputClasses` first creates a set `RESULT` which contains all class names in the repository. Next the `classRSF` clause is invoked and it recursively traverses `RESULT` and prints for each class an RSF tuple of **nodetype** *Class*.

```
outputClasses :-
        setof((X), class(X,_), RESULT),
        classRSF(RESULT).

classRSF( [X] | Xs) :-
        format("type ~p Class ~n", X).
classRSF( [] ).
```

Figure 7.4 Class fact to RSF clauses

Converting the individual patterns to RSF requires the traversal of the set RESULT which is produced by the `processTuples` clause in Figure 7.2. Each pattern instance is processed by the `outputPattern` clause shown in Figure 7.5. As with previous clause an RSF tuple is produced for each class encountered. The **nodetype** of the RSF tuple is

dependent on the role the class plays within the pattern.

```
outputPattern([(A,R,I) | Xs]) :-
      format("type ~p Abstraction~n",A),
      outputRefinedAbstractions(R),
      outputImplementors(A,I),
      outputPattern(Xs).
outputPattern([]).

outputRefinedAbstractions([R| Xs]) :-
      format(,"type ~p RefinedAbstraction~n",R),
      outputRefinedAbstractions(Xs).
outputRefinedAbstractions([]).

outputImplementors(A,[(I, C) | Xs]) :-
      format(,"type ~p Implementor~n",I),
      outputConcreteImplementors(C),
      outputImplementors(A,Xs).
outputImplementors(A,[]).

outputConcreteImplementors([C| Xs]) :-
      format(,"type ~p ConcreteImplementor~n",C),
      outputConcreteImplementors(Xs).
outputConcreteImplementors([]).
```

Figure 7.5 Bridge pattern to RSF tuple

All classes involved in a pattern produce two RSF tuples. One is of type *class,* and the other is role dependent. When RSF data is read by Rigi, it does not expect the same class name to be associated with two separate types. Fortunately, the most recent type read is the one adopted. It is therefore of importance to run the clause `outputClasses` before `outputBridges`.

The corresponding RSF formatting clauses for the remaining patterns and facts can be found in Appendix C and D, respectively.

7.4 Summary

This chapter described the steps required to recover instances of patterns from the fact database and produce the output for the visualization tool.

A two step strategy has been applied to identify unique instances of a pattern. First, tuples of classes satisfying the pattern clauses introduced in the previous chapter are extracted and added as new facts to the repository. Next, the tuples are consolidated according to a regular expression that defines exactly how the class tuples are combined to produce unique pattern instances.

In addition clauses responsible for generating input for the visualization tool have also been provided. These clauses reformat the fact database as well as the pattern instances into the visualization tool's input format.

A large number of clauses concerned with identifying, consolidating and formatting the desired design patterns have been introduced in this chapter. Figure 7.6 presents a single clause that combines all pattern processing steps into a single statement.

```
produceResults :-
        produceTuples,
        processTuples(RESULT),
        outputClasses,
        outputInheritance,
        outputPattern(RESULT).
```

Figure 7.6 Complete clause

When executed the clause produces instances of patterns by executing the clauses responsible for pattern consolidation. Next, the input for the visualization tool is produced by first generating class and inheritance RSF tuples and then producing those RSF tuples responsible for identifying the classes involved in the pattern.

8 A Case Study

The pattern detection strategy described in the previous chapters was tested on the public domain graphical user interface class library ET++. It is known to contain various occurences of design patterns in its implementation and it is frequently referenced as an example software project that was designed using patterns [1,2]. It is therefore an ideal candidate for a case study, guaranteeing the presence of patterns within the source code.

After the artifact extraction, the data repository is loaded into the Prolog runtime environment and primary and secondary processing is performed. The resulting RSF tuples are then passed as input to the visualization tool, where the information is rearranged to produce various representations of the extracted patterns.

This chapter presents the scope of the source code reverse engineered and the proportions of the resulting data repository. The form in which the resulting patterns are presented is discussed and, in addition to the views produced by the software engineering tool, a text based listing of the classes involved in a pattern is also provided. An attempt is made to reflect upon the benefits of using a visualization tool versus a text based representation. Observations are made using the visual representations that otherwise could not have been deduced from the textual representation.

8.1 Sample System

Table 8.1 is an overview of the system before and after the reverse engineering task has

been performed. The input section of the table describes the volume of the source code analyzed. The output section lists a count indicating how many of each type of language artifact were extracted to compose the domain knowledge.

The number of files given is a count of the files passed to the parser as input. Only the *cpp* files are passed and the *#include* directives are then resolved automatically. The actual number of files parsed is therefore somewhat larger than indicated. Similarly, the total lines of source code reflects a lower bound of the number of lines reverse engineered.

Input to parser	
Total Number of Source Code files	88
Total Number of Source Code Lines	28450
Output of parser	
Classes	344
Inheritance	271
Operations	4354
Calls	6107
Instantiations	685
References	1229

Table 8.1 ET++ system overview

Some of the original source code had to be modified, as the Brown University parser failed to parse certain C++ language constructs correctly. The constructs are primarily overloaded operators and certain type conversions. The modifications to the source code changed the syntax of the program but left the semantics unchanged.

A portion of 70% of the library source code is included in the test run. Attempts to parse more than that resulted in the occasional memory exception or abrupt termination of the source code parser. Parsing a large amount of source code at once results in a very

large AST, which may have led to memory problems or Stack overflows.

8.2 Presentation

The patterns extracted from the ET++ library are presented in two forms. One is tabular, where the rows represent the sets of classes extracted by the secondary processing steps. Each table represents the classes in the same way that they are organized according to the pattern *consolidation rules*. Classes in the same pattern are often in a 1-to-many relationship to one another and table cells are therefore split to accommodate several classes at once. Within each table individual pattern instances are separated by double lines.

An instance of a pattern often contains a large number of classes, and a textual representation often fails to present this information in a way that it is easily absorbed. It is therefore of interest to present large patterns visually, where the individual classes are arranged in such a way that their role in the pattern is revealed. The classes as well as the relationships connecting them to the pattern are presented.

Little has been done so far on analyzing the benefits of presenting the software engineer with a visual representation of an object oriented design pattern. The intent of this thesis is to provide an introduction to what the benefits can be and how they may impact software maintenance. The representations are restricted to a single type of pattern at a time, focussing primarily on the layout of the consolidated patterns, as well as the location of the patterns within the class hierarchy.

A design pattern governs much of the behavior of the classes involved. Derived classes consequently inherit that behavior. The lower in the class hierarchy the pattern occurs, the greater its influence. An assessment of the "spread" of a pattern is also possible when viewing it within the class hierarchy. Some patterns may be confined to a small portion of the inheritance tree, giving an indication that the pattern is restricted to a small portion of the system. Others may be spread among classes that are only remotely related if at all, giving an indication that the pattern may be involved in coupling or connecting entire

subsystems.

The visual representations consist of directed graphs that have been prepared using the Rigi software visualization system. Nodes within the graph represent classes and arcs represent relationships such as inheritance and reference. An arc is directed from the bottom of one node to the top of another. Many of the figures shown in this chapter exhibits screen snapshots of a Rigi application window [12].

8.3 Results

Occurrences of each of the four patterns (i.e. Adapter, Bridge, Composite and Command) were found within the source code of the ET++ system. An attempt is made to determine intentional or unintentional use of the patterns by analyzing the names of the classes within each occurrence. Class names that give an indication towards the role a class plays within a pattern are often a sign of an intentional use.

8.3.1 Adapter

The *consolidation rules* for the Adapter tuples produced ten unique instances, some of which have up to 24 *Clients*. Only a subset of Adapters found is present here, and a complete listing can be found in Appendix E. Table 8.2 presents five selected adapters, all of which have the *Target* and *Clients* in common. The names of the classes give no indication to an intentional use of the Adapter, but rather suggest an involvement of the classes in the Command pattern. As the Adapter pattern is a *subpattern* of the Command it may be the case that these classes will appear again within the Command pattern.

Target	Adapter	Adaptee	Client
Command	ChangedMarkerCommand	MarkerItem	BoundedCommandProcessor
			SingleCommandProcessor
Command	SplitMover	Box	BoundedCommandProcessor
			SingleCommandProcessor
Command	GraphNodeMover	VObject	BoundedCommandProcessor
			SingleCommandProcessor
Command	StretchWrapperStretcher	VObject	BoundedCommandProcessor
			SingleCommandProcessor
Command	ThumbMover	Slider	BoundedCommandProcessor
			SingleCommandProcessor

Table 8.2 Adapter patterns in ET++

Figure 8.2 shows a layout of the classes involved in the selected Adapter patterns. The

picture shows that the patterns overlap in the class named "Command" which performs the role of the *Target*. The Figure suggests the presence of only one Adapter which is composed of all the classes in the picture and a re-evaluation of the *consolidation rule* that produced the Adapters may be required.

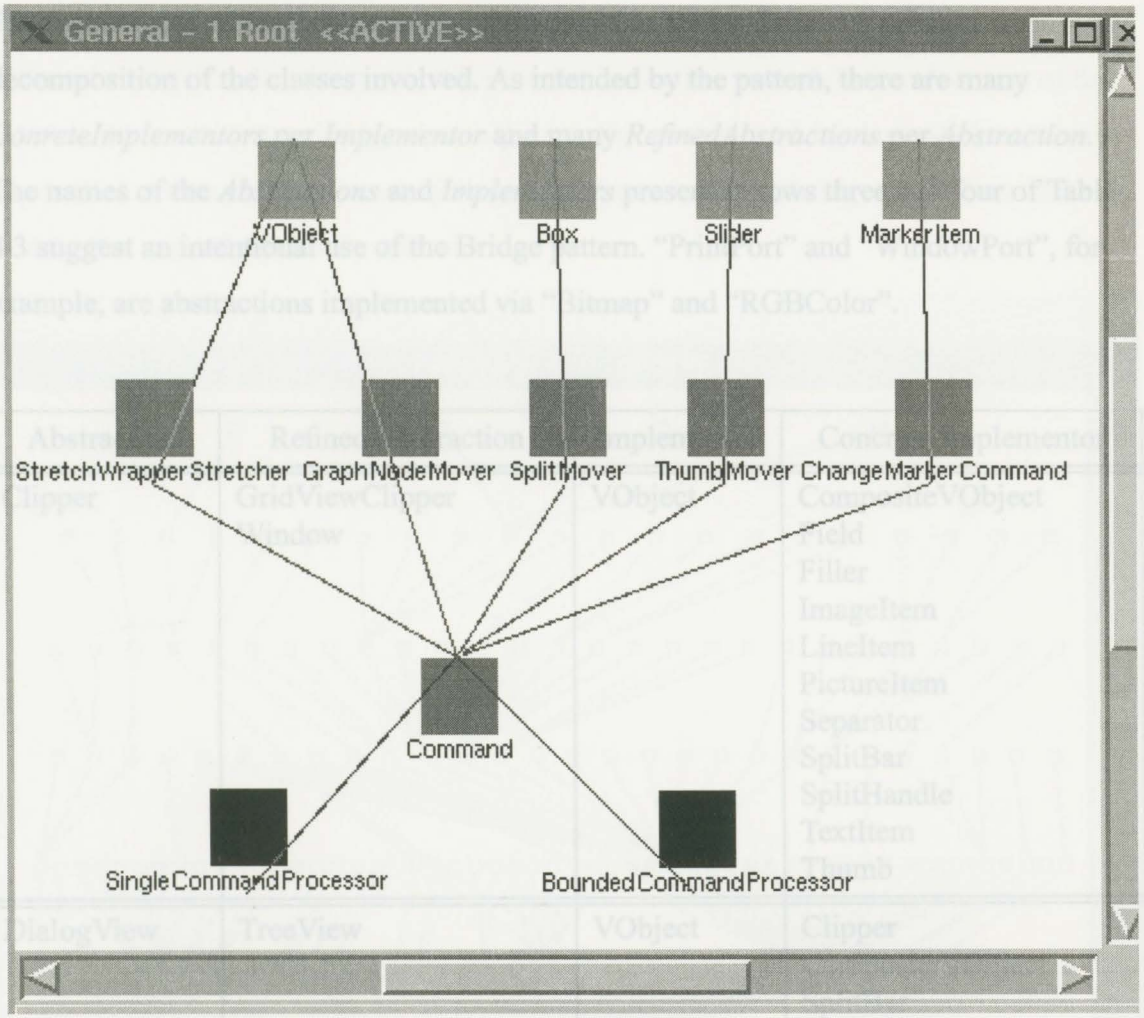

Figure 8.1 Adapter pattern layout A

The topmost row of classes represents the *Adaptees* and the second row the *Adapters*. The class named "Command" is the *Target*, and the two *Clients* are shown on the bottom of the picture. Different textured arcs represent the inheritance and reference relationships present among the classes in the patterns. Both of the *Adapters* named

"StretchWrapperStretcher" and "GraphNodeMover" adapt the same class "VObject" to the interface defined by the *Command*.

8.3.2 Bridge

Four occurances of the Bridge pattern were found in ET++. Table 8.3 presents a decomposition of the classes involved. As intended by the pattern, there are many *ConreteImplementors* per *Implementor* and many *RefinedAbstractions* per *Abstraction*. The names of the *Abstractions* and *Implementors* present in rows three and four of Table 8.3 suggest an intentional use of the Bridge pattern. "PrintPort" and "WindowPort", for example, are abstractions implemented via "Bitmap" and "RGBColor".

Abstraction	Refined Abstraction	Implementor	Concrete Implementor
Clipper	GridViewClipper Window	VObject	CompositeVObject Field Filler ImageItem LineItem PictureItem Separator SplitBar SplitHandle TextItem Thumb
DialogView	TreeView	VObject	Clipper CompositeVObject SplitBar View
Port	PrintPort WindowPort	Ink	Bitmap RGBColor
StaticTextView	TextView	Text	CheapText GapText
		TextPager	LinePager

Table 8.3 Bridge patterns in ET++

Figure 8.2 shows the Bridges within the class hierarchy. The coloring of the classes in the pattern becomes darker in the order of *Abstraction*, *RefinedAbstraction*, *Implementor* and *ConcreteImplementor*. *Abstractions* are displayed in the lightest shade of grey and *ConcreteImplementors* in the darkest shade. Although very cluttered with arcs and different colored nodes, the graph still gives an indication toward the involvement of the patterns. One occurrence bridges two adjacent *Abstractions* and *Implementations* where *Implementor* and *Abstraction* are siblings. Another occurrence bridges two parts of the system that are only remotely related.

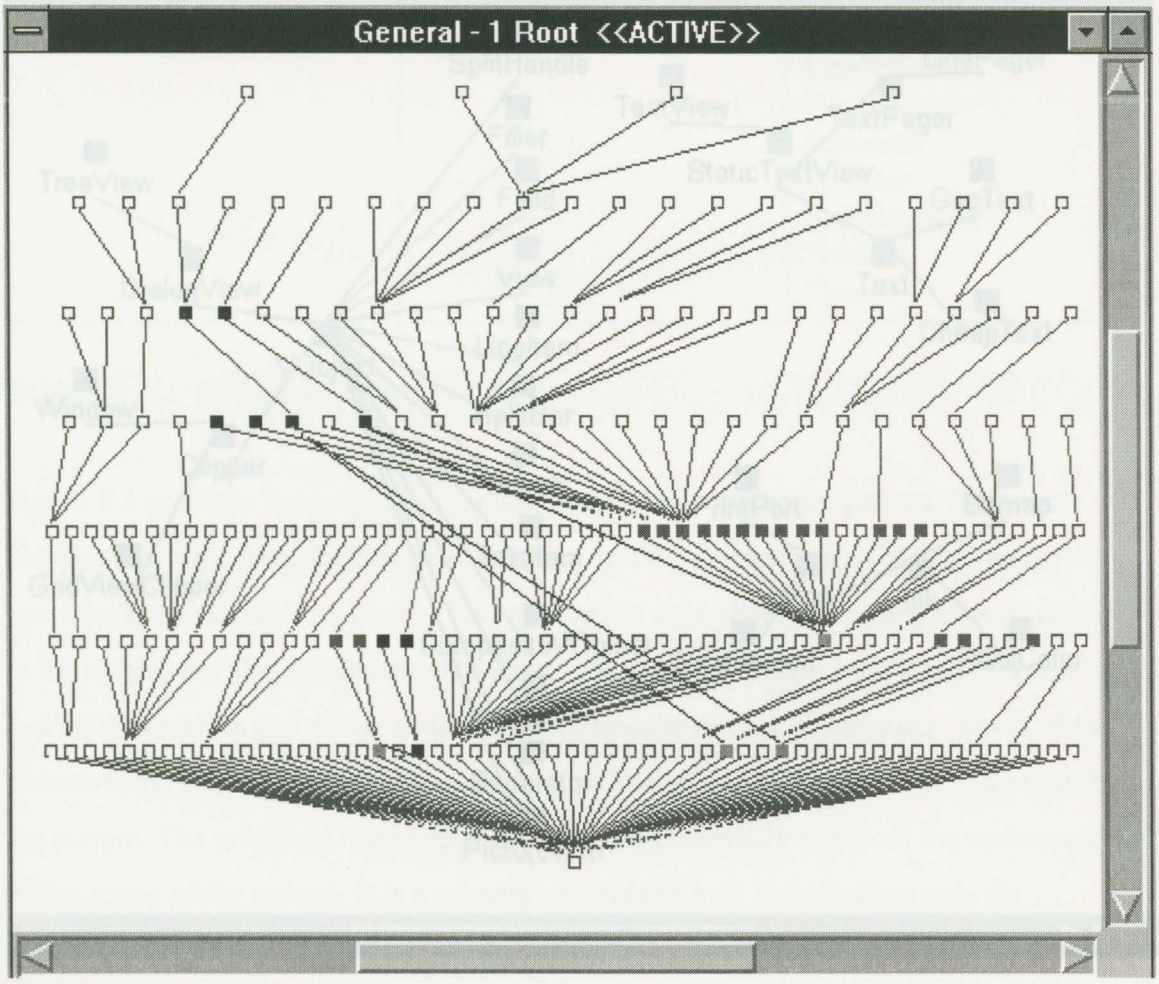

Figure 8.2 Bridge pattern layout A

Figure 8.3 provides another, less busy view of the patterns. The graph is a representation of all the classes listed in Table 8.3. The classes are arranged in such a way as to separate the individual occurences of the Bridge. Note that there are only three distinct groupings of classes suggesting the presence of only three Bridges and not four as the data in Table 8.3 indicates. Rows one and two of Table 8.3 describe two instances with identical *Implementor* called "VObject" and two separate *Abstractions*, named "Clipper" and "DialogView". The Bridges intersect in the *Implementor* class causing it to appear as a single Bridge rather than two, as generated by the *consolidation rules*. Rows three and four of Table 8.3 are represented by the classes in the upper and lower right hand side of the screen, respectively.

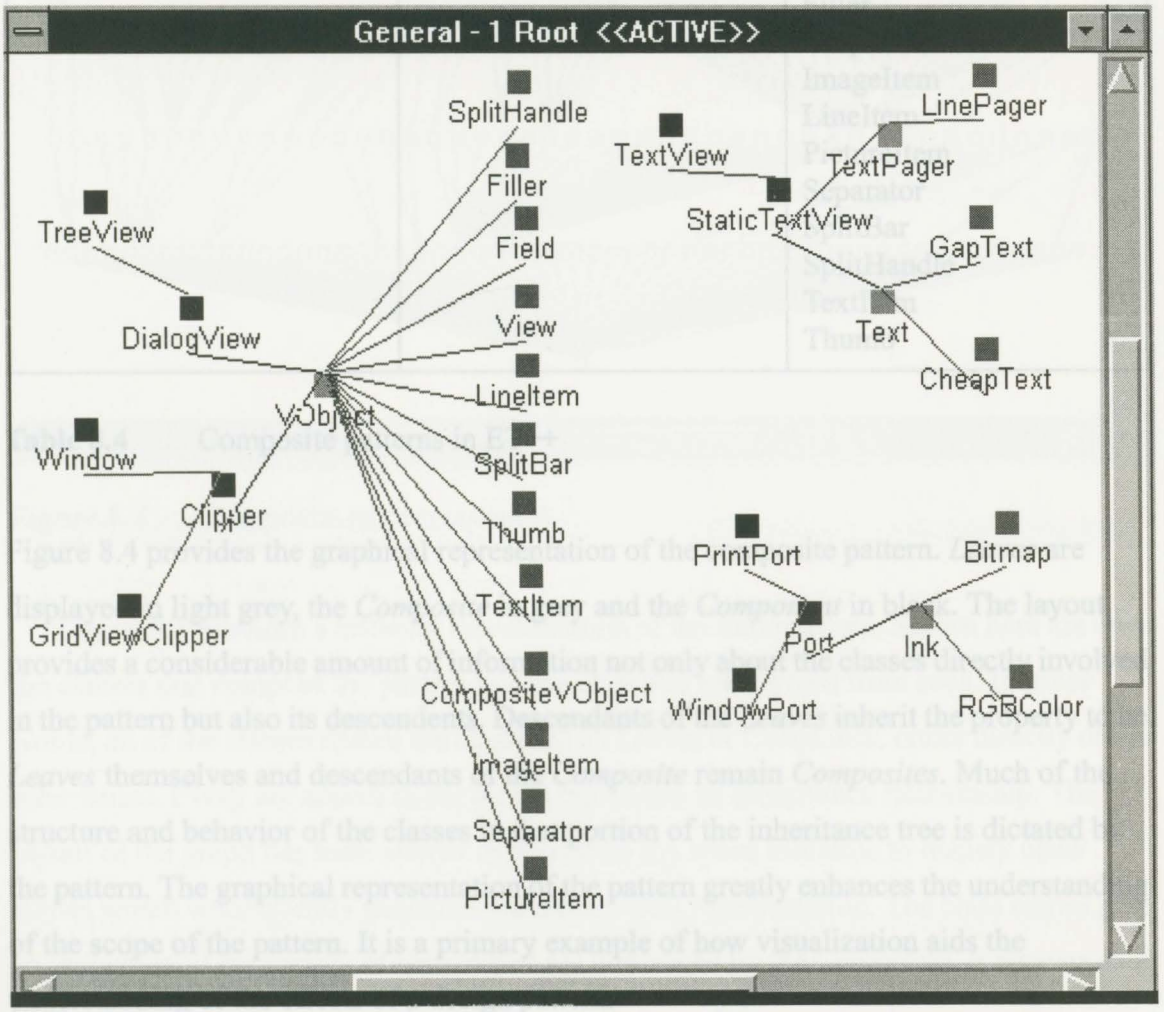

Figure 8.3 Bridge pattern layout B

8.3.3 Composite

Only one occurrence of the Composite pattern has been detected in the ET++ source code. The name of the *Composite* class, "VCompositeObject", gives a clear indication that the pattern has intentionally been used in the design. A large number of leaves have been detected for this Composite, suggesting the influence of the pattern to be considerable.

Component	Composite	Leaf
VObject	VCompositeObject	Clipper Field Filler GraphPath ImageItem LineItem PictureItem Separator SplitBar SplitHandle TextItem Thumb

Table 8.4 Composite patterns in ET++

Figure 8.4 provides the graphical representation of the composite pattern. *Leaves* are displayed in light grey, the *Composite* in grey and the *Component* in black. The layout provides a considerable amount of information not only about the classes directly involved in the pattern but also its descendents. Descendants of the *Leaves* inherit the property to be *Leaves* themselves and descendants of the *Composite* remain *Composites*. Much of the structure and behavior of the classes in that portion of the inheritance tree is dictated by the pattern. The graphical representation of the pattern greatly enhances the understanding of the scope of the pattern. It is a primary example of how visualization aids the understanding of the effects of a design pattern.

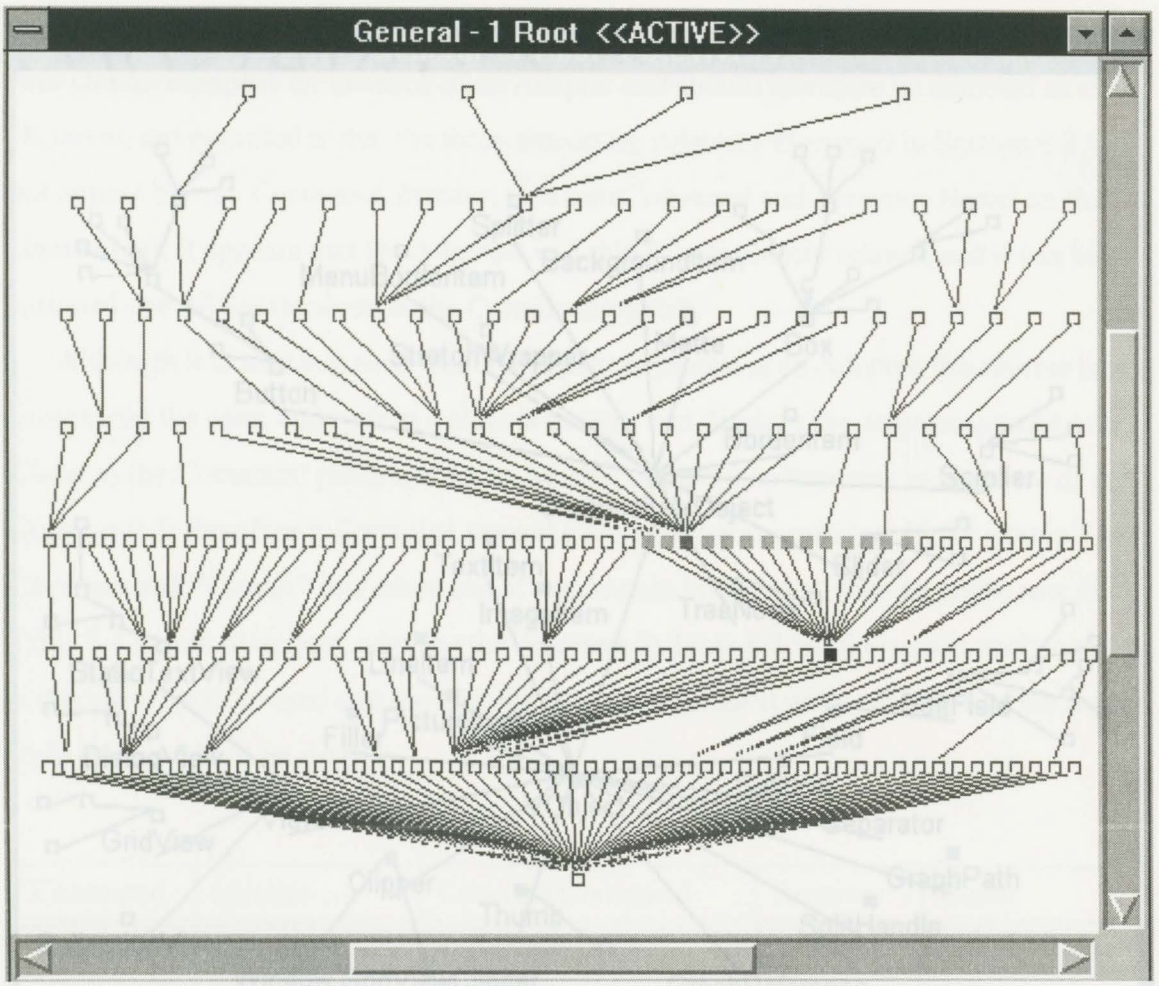

Figure 8.4 Composite pattern layout A

Figure 8.5 provides a different representation of the same pattern. Shown here are only the classes that compose the pattern and the ones that are derived from such. In other words, all of the classes shown are classified as *Leaves* or *Composite*, either directly or via inheritance. Every arc shown in the layout represents an inheritance relationship. The layout of the graph has been altered to maximize the space available to display class names which consequently sacrifices the hierarchical representation. The class names give an indication of the variety of classes that are influenced by the Composite. It appears as if the pattern is mainly concerned with the organization of the graphical user interface classes.

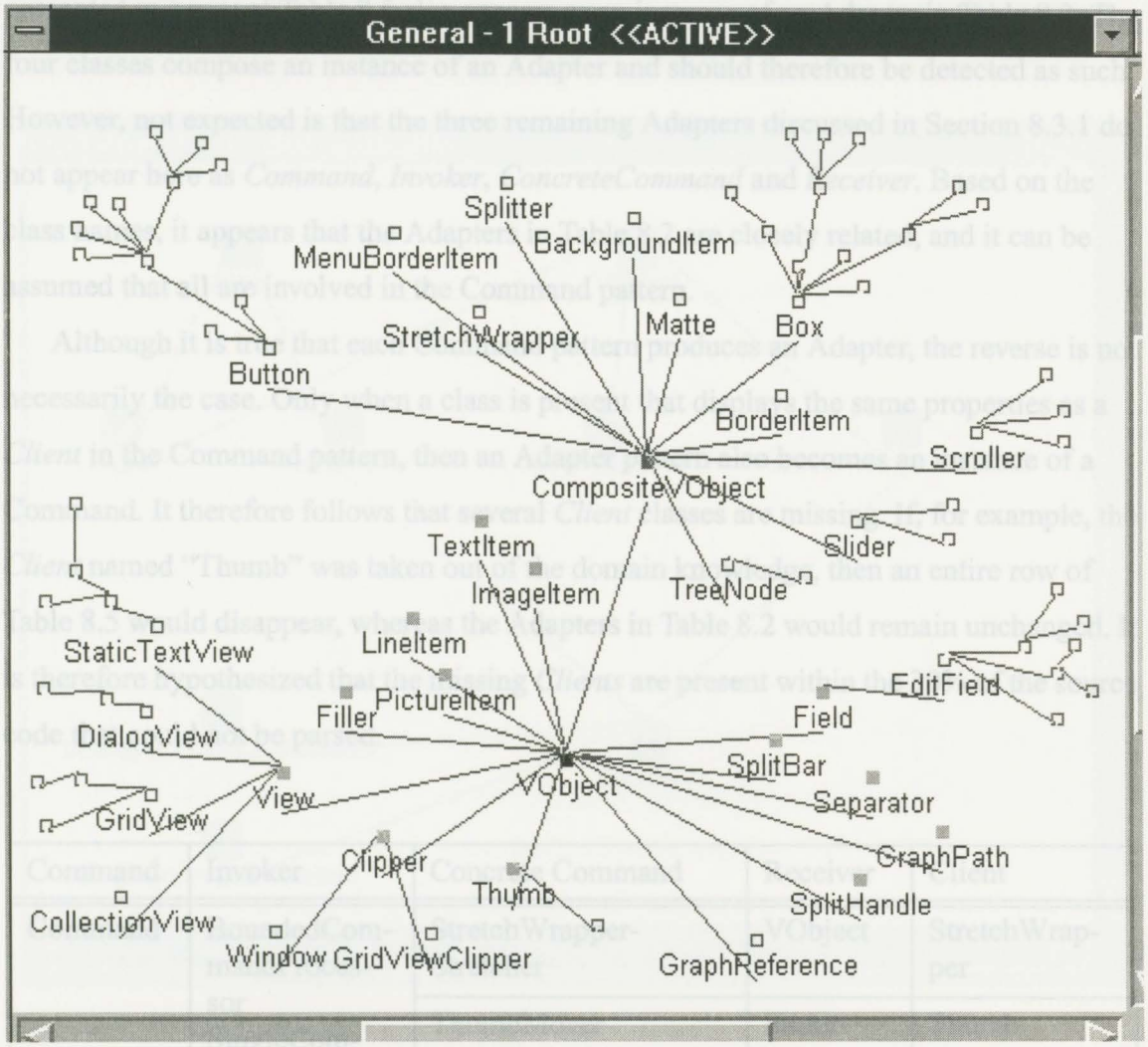

Figure 8.5 Composite pattern layout B

8.3.4 Command

ET++ contains one occurrence of the Command pattern. There are two *ConcreteCommands*, each of which is used by separate *Clients* and has a separate *Receiver*. The naming of the classes involved in the pattern again gives an indication of an intentional use of the pattern, which is reflected by the names "Command", "BoundedCommandProcessor" and "SingleCommandProcessor".

As expected, each of the *Command*, *Invoker*, *ConcreteCommand* and *Receiver* tuples presented in a row of Table 8.5 also appears as an instance of an Adapter in Table 8.2. The four classes compose an instance of an Adapter and should therefore be detected as such. However, not expected is that the three remaining Adapters discussed in Section 8.3.1 do not appear here as *Command*, *Invoker*, *ConcreteCommand* and *Receiver*. Based on the class names, it appears that the Adapters in Table 8.2 are closely related, and it can be assumed that all are involved in the Command pattern.

Although it is true that each Command pattern produces an Adapter, the reverse is not necessarily the case. Only when a class is present that displays the same properties as a *Client* in the Command pattern, then an Adapter pattern also becomes an instance of a Command. It therefore follows that several *Client* classes are missing. If, for example, the *Client* named "Thumb" was taken out of the domain knowledge, then an entire row of Table 8.5 would disappear, whereas the Adapters in Table 8.2 would remain unchanged. It is therefore hypothesized that the missing *Clients* are present within the 30% of the source code that could not be parsed.

Command	Invoker	Concrete Command	Receiver	Client
Command	BoundedCommandProcessor SingleCommandProcessor	StretchWrapperStretcher	VObject	StretchWrapper
		ThumbMover	Slider	Thumb

Table 8.5 Command patterns in ET++

Figure 8.6 presents the Command pattern visually. The classes are arranged in such a way that the *consolidation rule* used to compose the instance of the pattern, is emphasized. The triplets of classes arranged in the upper left and right hand side of the picture correspond to {*ConcreteCommand*, {*Receiver*, *Clients**}*}. The class in the center corresponds to the *Command* and the two classes in the lower left corner to *Invoker* *. All references,

inheritance and instantiations are represented by different colored arcs. The arc from class "Thumb" to the class "VObject" denotes an inheritance that is not related to the pattern.

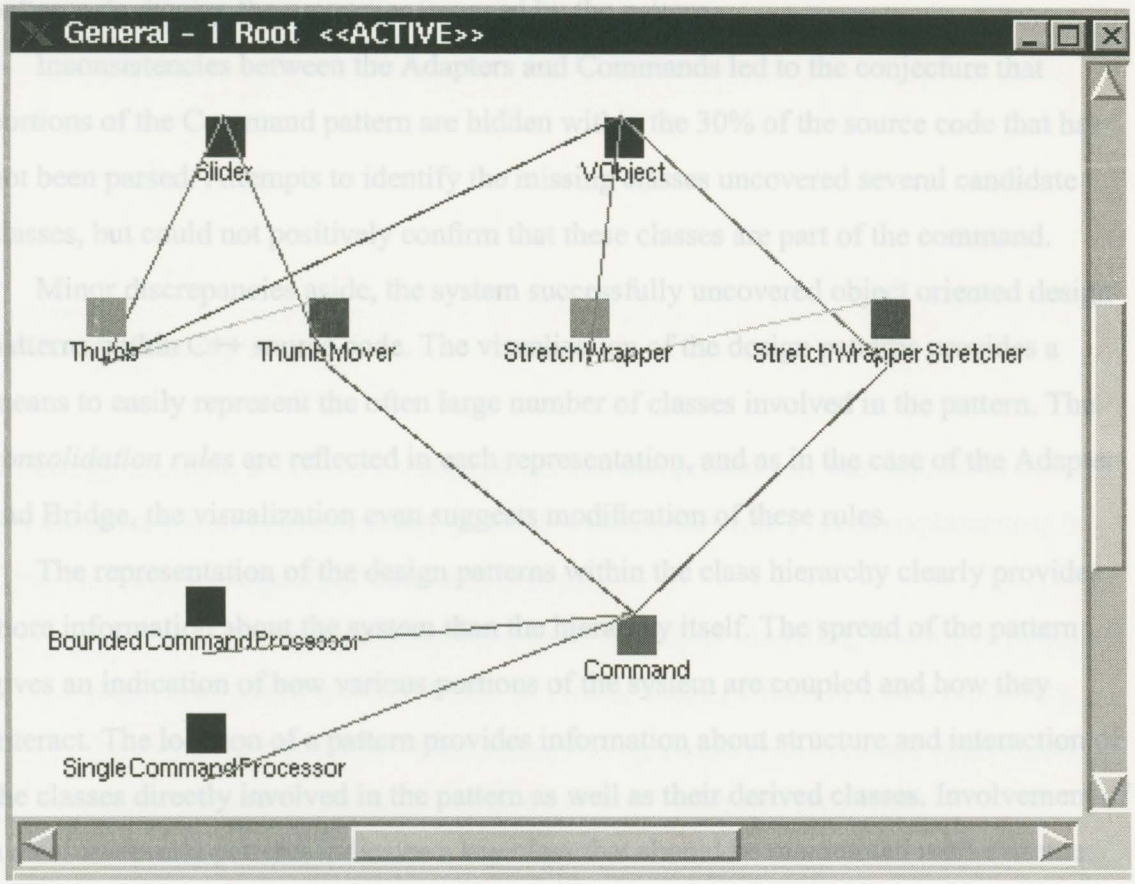

Figure 8.6 Command pattern layout A

8.4 Discussion

By reverse engineering the ET++ library a variety of patterns were uncovered. Some are clearly present due to designer's intent, which was concluded by a correlation of the class names with their roles in the pattern. Others appear to have been applied unintentionally.

Confirmation of the actual presence of the patterns is not provided. It has been verified

that the system introduced here correctly parses the source into its OMT constructs and correctly extracts these constructs via the querying mechanisms. If the strategies and mechanisms applied are correct then so are the results. All of the classes identified as patterns do display the properties imposed by the pattern.

Inconsistencies between the Adapters and Commands led to the conjecture that portions of the Command pattern are hidden within the 30% of the source code that has not been parsed. Attempts to identify the missing classes uncovered several candidate classes, but could not positively confirm that these classes are part of the command.

Minor discrepancies aside, the system successfully uncovered object oriented design patterns within C++ source code. The visualization of the design patterns provides a means to easily represent the often large number of classes involved in the pattern. The *consolidation rules* are reflected in each representation, and as in the case of the Adapter and Bridge, the visualization even suggests modification of these rules.

The representation of the design patterns within the class hierarchy clearly provides more information about the system than the hierarchy itself. The spread of the pattern gives an indication of how various portions of the system are coupled and how they interact. The location of a pattern provides information about structure and interaction of the classes directly involved in the pattern as well as their derived classes. Involvement of a class in several pattern, indicates a key class that should be maintained with extreme care.

9 Conclusions

9.1 Summary

A strategy common to software visualization and comprehension was applied to detect and visualize object oriented design patterns within a software system implemented in C++. The strategy consists of three major steps.

- Reverse engineering of source code into its Object Modeling Technique (OMT) constructs, which are then stored as Prolog language facts in a data repository.
- Querying the repository by executing Prolog clauses which model object oriented design patterns in terms of Object Modeling Technique constructs.
- Visualization of the design patterns by representing classes as nodes and relationships as arcs, using the Rigi system.

The logic programming language Prolog acts as the formalism with which to represent the domain knowledge and define design pattern queries and processing rules. The OMT constructs provide the data model governing how the domain knowledge and pattern queries are formatted. Rigi allows for a visual representation of the design patterns embedded in the class hierarchy. Input for the visualization tool is generated by converting the information present in the data repository into an intermediate format called RSF (Rigi Standard Format).

The system leverages and combines a variety of methodologies, technologies and tools of the fields of software engineering, logic programming, object oriented design, compiler

construction and design patterns.

A pattern-rich source code sample was used to demonstrate and test the introduced pattern recognition strategy. As a result, several Adapter, Composite, Command and Bridge patterns were identified. Minor discrepancies among the Command and Adapter patterns may be caused by incomplete source code parsing.

9.2 Observations

This section discusses several noteworthy observations made by the author during the course of this thesis.

9.2.1 Inconsistent Pattern Specifications

The specification of the selected patterns was found to be vague at times. Classes that are specified as concrete may be abstract as well. Associations and aggregations are mostly interchangeable with little or no impact on the intent and meaning of the pattern. In the case of the Composite pattern an association was implemented via a container class. This resulted in an instance of the pattern that considerably deviated from the specifications given by Gamma, *et al.* [2], but which was nevertheless a Composite. Some patterns such as the Bridge, introduce two classes with the same properties, to indicate that a class can be involved in a pattern several times. *ConcreteImplementorA* and *ConreteImplementorB* play exactly the same role in the pattern. In the case of the Composite pattern, however, there is only one *Leaf* class, although the presence of many *Leaves* is expected [2].

However, since design patterns are architectural examples, varying interpretation and implementation are to be expected.

9.2.2 Nested Patterns

Closer examination of the Command pattern shows that some classes involved in the

pattern display all the properties of an Adapter pattern. In other words there are patterns which are extensions of others and they can hence be modeled in terms of such. A nesting of patterns where possible will not necessarily simplify the understanding of the pattern, but it will certainly simplify the querying. The pattern search can be performed in a smallest to largest manner, where the simple patterns are found first and added to the repository. Complex patterns are then modeled in terms of the smaller ones.

9.2.3 Extensible and Flexible Querying

The querying mechanism applied is easily extended to find other patterns or complex software abstractions. An OMT diagram is with little effort translated into a Prolog clause. Prolog furthermore provides many of the features of conventional programming languages such as I/O primitives, iterative control structures and debugging. These features allow the software engineer to customize the queries and data repository far beyond the applications introduced in this thesis.

9.3 Conclusion

This thesis presents a strategy that was successfully applied to uncover and visualize instances of object oriented design patterns from C++ source code. It combines methodologies inherent to reverse engineering and software visualization, to provide a tool which allows the software engineer to detect any structural object oriented design pattern and consequently gather valuable information about a system.

Reverse engineering OMT constructs provides an easy to use and flexible basis for query design and implementation. Object oriented software design patterns, which are generally specified using the OMT formalism are easily modeled and translated into queries. However, the pattern *consolidation rules* introduced in section 7.2 turned out to be non-trivial. Derivation and especially implementation of these rules is complicated at best. Unfortunately, it appears that these consolidation rules are required for every pattern,

making the entire process of pattern recovery more cumbersome than initially expected.

The visualization of the design patterns within the reverse engineered software artifacts greatly improves comprehension and analysis of the extracted pattern. Several benefits became apparent:

- A visual representation of a design pattern provides an easy way to absorb the structure of the pattern itself. The *consolidation rules* applied here can produce very large patterns, for which a textual representation becomes infeasible.
- Visualization within the class hiearchy reveals behavioral and structural aspects of many of the classes within the inheritance tree. Every class derived from one involved in a pattern automatically inherits the behavior imposed by the pattern.
- Object oriented design patterns document the system.
- The spread of the pattern in the class hierarchy gives an indication to how various portions of the system are coupled and how they interact.
- Subsystem decomposition based on design pattern becomes possible. The patterns also provide a commonly accepted terminology.

Given an observer knowledgeable of design pattern, the amount of additional information that has to be absorbed by the observer is minimal compared to the amount of information gained. A mere annotation of the class hierarchy is sufficient to present an observer with all the occurrences of design patterns within a subject system. The information encapsulated by a pattern is by far richer than the information contained within the class hiearchy in which it is embedded.

However, several issues with respect to the implemetation of the strategy still have to be resolved. Design patterns are referred to as architectural examples, allowing for varying interpretation and especially implemetation. It is important to recognize those parts of a pattern that are open for variations and design the queries accordingly. Performance limits of the Prolog clauses have been reached with a system the size of the ET++ library and a more effective querying mechanisms is required.

In retrospect, this paper should have been focused on "searching for complex abstractions" within the source code, which made this work independent of the term pattern. Patterns are simply one type of complex software abstraction. The strategy

introduced here however, is suited to detect any conceivable object model.

9.4 Suggested Research

Following are some of the author's questions and ideas that arose throughout the completion of this work.

9.4.1 Pattern query catalog

A complete catalog of queries and processing clauses suitable for various design pattern is required in order to provide a complete reverse engineering framework. A large number of structural design patterns exist, and for each a complete set of clauses concerning identification and processing is needed. A catalog of queries will provide the software engineer with a powerful information gathering tool set.

9.4.2 Software Architecture Detection

The strategy introduced here can be scaled to identify higher level software abstractions such as software architectures. They are often specified in terms of interacting design patterns [14],[15]. A basis for more advanced queries is built by adding a new fact to the repository for each instance of a pattern found. New queries can then be designed to detect particular arrangements of design patterns.

9.4.3 Brute Force Design Pattern Discovery

The invention of new design patterns has turned into a sport among computer practitioners. To stay ahead in the game one may take an automated approach to find new patterns. Large quantities of source code are reverse engineered to built a sufficiently large repository. A powerful search engine then uncovers frequently recurring arrangements of

programming language artifacts. The most frequent arrangements may describe an undiscovered design pattern. This strategy can also be scaled to detected frequently used arrangements of design patterns. However, such a task requires an immense software repository and an extremely powerful search engine.

9.4.4 Automated Design Verification

System designers are often faced with verifying whether an implemetor's work actually conforms to the specified design. Source code inspections, for example, are one approach to verify implementations. The strategy introduced here may provide another option to verify the design, by translating the OMT specifications into Prolog clauses and then querying the repository generated from the source code. If all clauses are satisfied then the design is present within the source code.

9.4.5 Search Engine Design by Prototyping

Although the Prolog language provides the flexibility to design various types of queries and processing clauses, it considerably lacks in runtime performance. The performance required to analyze large scale systems can not be achieved using Prolog. This bottleneck is to a large extent caused by the multitude of supported language features and the fact that Prolog is an interpreted language. However, Prolog can be used to design and test prototype search engines. Queries and processing clauses can be designed, tested and improved, until a querying framework is defined. A tailor made search engine can then be implemented using a faster imperative programming language, which optimizes searches and removes all unnecessary features that otherwise slow down performance.

9.4.6 Real-time Queries

Another possibly useful feature may be to present a query result in real time. Allowing the software engineer to change queries on the fly may greatly increase the effectiveness of

the information gathering process. Queries could quickly be revised and the effects immediately be observed. A visual querying tool based on an OMT editor would allow the software engineer to modify queries using an *point and click* approach. A search engine running in the background then constantly produces the result set.

Appendices

A Primary Processing Clauses

Adapter:

```
produceTuples :-
     setof((W,X,Y,Z),adapter(W,X,Y,Z),RESULT),
     addTuples(RESULT).

addTuples((W,X,Y,Z) | Xs) :-
     assert(adapterTuple(W,X,Y,Z)),

addTuples([]).
```

Bridge:

```
produceTuples :-
     setof((W,X,Y,Z),bridge(W,X,Y,Z),RESULT),
     addTuples(RESULT).

addTuples((W,X,Y,Z)) | Xs) :-
     assert(bridgeTuple(W,X,Y,Z)),

addTuples([]).
```

Composite

```
produceTuples :-
     setof(     (A,B,C,D,E),
                component(A,B,C,D,E),
                RESULTA
          ),
     addComponents(RESULTA),
     setof((F,G),composite(F,G),RESULTB),
     addComposites(RESULTB),
     setof((H,I),leaf(H,I),RESULTC),
     addLeafs(RESULTC).

addComponents([(A,B,C,D,E) | Xs]) :-
     assert(componentTuple(A,B,C,D,E)),
     addComponents(Xs).
addComponents([]).

addComposites([(A,B) | Xs]) :-
     assert(compositeTuple(A,B)),
     addComposites(Xs).
addComposites([]).

addLeafs([(A,B) | Xs]) :-
     assert(leafTuple(A,B)),
     addLeafs(Xs).
addLeafs([]).
```

Command

```
produceTuples :-
     setof((V,W,X,Y,Z),command(V,W,X,Y,Z),RESULT),
     addTuples(RESULT).

addTuples((V,W,X,Y,Z) | Xs) :-
     assert(commandTuple(V,W,X,Y,Z)),
addTuples([]).
```

B Pattern Consolidation Clauses

Adapter

```
processTuples(RESULT) :-
     setof(
          (T,A,E,C),
          (
               adapters(T,A,E),
               clients(T,A,E,C)
          ),
     RESULT
).

adapters(T,A,E) :- adapterTuple(_,T,A,E).

client(T,A,E,C) :- adapterTuple(C,T,A,E).
clients(T,A,E,RESULT) :-
     setof(
          R,
          client(T,A,E,R),
          RESULT
     ).
```

Bridge

```
processTuples(RESULT) :-
     setof(
          (A,R,I),
          (
               implementors(A,I),
               refinedAbstractions(A,R)
          ),
          RESULT
     ).

refinedAbstraction(A,R) :- bridgeTuple(A,R,_,_).
refinedAbstractions(A,RESULT) :-
     setof(
          R,
          refinedAbstraction(A,R),
          RESULT
     ).
```

```
implementor(A,I) :- bridgeTuple(A,_,I,_).
implementors(A,RESULT) :-
     setof((I,C),
     (
          implementor(A,I),
          concreteImplementors(A,I,C)
     ),
     RESULT
     ).
concreteImplementor(A,I,C):- bridgeTuple(A,_,I,C).
concreteImplementors(A,I,RESULT) :-
     setof(
          C,
          concreteImplementor(A,I,C),
          RESULT
     ).
```

Composite

```
processTuples(RESULT) :-
     setof(
          (C,O,L),
          (
               composites(C,O),
               leafs(C,L)
          ),
          RESULT
     ).

composites(C,RESULT) :-
     setof(
          O,
          compositeTuple(C,O),
          RESULT
          ).
```

```
leafs(C,RESULT) :-
     setof(
          L,
          leafTuple(C,L),
          RESULT
     ).
```

Command

```
processTuples(RESULT) :-
setof(
     (C,Cm,I),
     (
          command(C),
          ccommands(C,Cm),
          invokers(C,I)
     ),
     RESULT
).


command(C) :- commandTuple(_,_,_,C,_).

invoker(C,I) :- commandTuple(_,_,_,C,I).
invokers(C,RESULT) :-
     setof(
          I,
          invoker(C,I),
          RESULT
     ).


client(C,Cm,Rc,Cl) :- commandTuple(Cl,Rc,Cm,C,_).
clients(C,Cm,Rc,RESULT) :-
     setof(
          Cl,
          client(C,Cm,Rc,Cl),
          RESULT
     ).

receiver(C,Cm,Rc) :- commandTuple(_,Rc,Cm,C,_).

ccommand(C,Cm) :- commandTuple(_,_,Cm,C,_).
```

```
ccommands(C,RESULT) :-
        setof(
                (Cm,RC),
                (
                        ccommand(C,Cm),
                        setof(
                                (Rc,Cl),
                                (
                                        receiver(C,Cm,Rc),
                                        clients(C,Cm,Rc,Cl)
                                ),
                                RC
                        )
                ),
                RESULT
        ).
```

C Pattern to RSF Conversion Clauses

Adapter

```
outputPatterns([(T,A,E,C) | Xs]) :-
     format("type ~p Target~n",T),
     format("type ~p Adapter~n",A),
     format("type ~p Adaptee~n",E),
     printClients(C,T),
     outputPatterns(Xs).
outputPatterns([]).

printClients([R | Xs], T) :-
     format("type ~p Client~n",R),
     printClients(Xs,T).
printClients([],T).
```

Bridge

```
outputPatterna([(A,R,I) | Xs]) :-
     format("type ~p Abstraction~n",A),
     outputRefinedAbstractions(R),
     outputImplementors(A,I),
     outputPattern(Xs).
outputPattern([]).

outputRefinedAbstractions([R| Xs]) :-
     format("type ~p RefinedAbstraction~n",R),
     outputRefinedAbstractions(Xs).
outputRefinedAbstractions([]).

outputImplementors(A,[(I, C) | Xs]) :-
     format("type ~p Implementor~n",I),
     outputConcreteImplementors(C),
     outputImplementors(A,Xs).
outputImplementors(A,[]).

outputConcreteImplementors([C| Xs]) :-
     format("type ~p ConcreteImplementor~n",C),
     outputConcreteImplementors(Xs).
outputConcreteImplementors([]).
```

Composite

```
outputPatterns([(C,O,L) | Xs], ,) :-
     format("type ~p Component~n",C),
     printComposites(O),
     printLeafs(L),
     outputPatterns(Xs).
outputPatterns([]).

printComposites([O | Xs]) :-
     format("type ~p Composite~n",O),
     printComposites(Xs).
printComposites([]).

printLeafs([L | Xs]) :-
     format("type ~p Leaf~n",L),
     printLeafs(Xs).
printLeafs([]).
```

Command

```
outputPatterns([(C, Cm, I) | Xs]) :-
     format("type ~p Command~n",C),
     printCCommands(Cm),
     printInvokers(C,I),
     outputPatterns(Xs).
outputPatterns([]).

printInvokers(C,[I | Xs]) :-
     format("type ~p Invoker~n",I),
     printInvokers(C,Xs).
printInvokers(C,[]).

printClients(Cm,R,[C | Xs]) :-
     format("type ~p Client~n",C),
     printClients(Cm,R,Xs).
printClients(Cm,R,[]).

printReceivers(Cm,[(R,C) | Xs]):-
     format("type ~p Receiver~n",R),
     printClients(Cm,R,C),
     printReceivers(Cm,Xs).
printReceivers(Cm,[]).

printCCommands([(Cm,RC) | Xs]) :-
```

```
        format("type ~p ConcreteCommand~n",Cm),
        printReceivers(Cm,RC),
        printCCommands(Xs).
printCCommands([]).
```

D Facts to RSF Conversion Clauses

Class

```
outputClasses :-
     setof((X), class(X,_), RESULT),
     classRSF(RESULT).

classRSF( (X) | Xs) :-
     format("type ~p Class ~n", X).
classRSF( [] ).
```

Inheritance

```
outputInheritance :-
     setof((X,Y), inheritance(X,Y), RESULT),
     inheritanceRSF(RESULT).

inheritanceRSF( (X,Y) | Xs) :-
     format("inheritance ~p ~p~n", [X,Y]).
inheritanceRSF( [] ).
```

Instantiation

```
outputInstantiation :-
     setof((X,Y), instantiation(X,Y), RESULT),
     instantiationRSF(RESULT).

instantiationRSF( (X,Y) | Xs) :-
     format("instantiation ~p ~p~n", [X,Y]).
instantiationRSF( [] ).
```

Reference

```
outputRefernce :-
     setof((X,Y),
          (
               association(X,Y,_);
               aggregation(X,Y,_)
          ),
          RESULT
     ),
     referenceRSF(RESULT).
```

```
referenceRSF( (X,Y) | Xs) :-
    format("reference ~p ~p~n", [X,Y]).
classRSF( [] ).
```

E ET++ Adapter Patterns

Target	Adapter	Adaptee	Client
Command	ChangedMarkerCommand	MarkerItem	BoundedCommandProcessor SingleCommandProcessor
Command	SplitMover	Box	BoundedCommandProcessor SingleCommandProcessor
Command	GraphNodeMover	VObject	BoundedCommandProcessor SingleCommandProcessor
Command	StretchWrapperStretcher	VObject	BoundedCommandProcessor SingleCommandProcessor
Command	ThumbMover	Slider	BoundedCommandProcessor SingleCommandProcessor
CommandProcessor	SingleCommandProcesso	Command	Manager
Ink	RGBColor	Port	BackgroundItem CellAttributes CharDesc CharStyleSpec Clipper Filler GraphPath GraphReference Look PictPort Port TextReader

Target	Adapter	Adaptee	Client
TextView	ShellTextView	Mark	CharStyleCommand DragAndDropSelector EditDialog FindDialog MoveTextCommand ParaStyleCommand TextCommand TextRangeSelector TextViewOverlay TypingTimer VObjectText
VObject	View	Clipper	CellSelector ChangeDialog Clipper CollectionView DialogView FindDialog FindFocus GraphNodeMover Menu MenuBar MenuItemSelector MouseTracker PrintManager ReferenceMark Scroller SplitMover Splitter StretchWrapper StretchWrapper-Stretcher TreeView VObject VObjectCommand VObjectMark Window
VisualMark	VObjectMark	VObject	PasteVObjectCommand

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides,"*Design Patterns Abstraction and reuse of object-oriented Design,*" In European Conference on Object-oriented Programming, Kaiserslautern, Germany, July 1993. Published as Lecture notes in Computer Science #707, pp. 406-431, Springer-Verlag, 1993.

[2] Erich Gamma, Richard Helm, Ralph Johnson,John Vlissides,"*Design Patterns : Elements of Reusable Object-Oriented Software*", Addison-Wesley Publishing Company, 1995.

[3] James O. Coplien, "*Progress on Patterns: Highlights of PloP/94*", Proceedings of Object Expo Europe, September 26-30, 1994.

[4] James O. Coplien, "*Software Design Patterns: Common Questions and Answers*", Software Production Research Department, AT&T Bell Laboratories.

[5] John Vlissides, "*Reverse Architecture*", Technical Report, Dagstuhl Seminar 9508

[6] Alberto Mendelzon, "*Reverse Engineering by Visualization and Querying*", Software Concepts and Tool 16, pp. 170-182, 1995.

[7] Christian Kraemer, "*Design Recovery by Automated Search for Structural Design Patterns in Object Oriented Software*", Working Conference on Reverse Engineering, Monterey, November 1996.

[8] Kent Beck, Ron Cocker, James O. Coplien, Lutz Dominick, Gerad Meszaros, Frances Paulisch, John Vlissides, "*Industrial Experience with Design Patterns*", Proceedings of the 18th International Conference on Software Engineering, pp. 103-114, IEEE Computer Society Press, March 1996.

[9] Thomas Ball, Stephen G. Eick, "*Software Visualization in the Large*" Computer, 29(4), pp. 33-43, IEEE journal Computer, April 1996.

[10] Leon Sterling, Ehud Shapiro, "*The Art of Prolog*", MIT Press , 1986

[11] Aho, Sethi and Ullman's "*Compilers - Principles, Techniques and Tools*", 2nd edition, Addison-Wesley Publishing Company, 1994.

[12] Hausi A. Mueller, Mehemt A. Orgun, Scott R Tilley, James Uhl, "*A Reverse-engineering Approach to Subsystem Structure Identification*", Software Maintenance: Research and Practice, VOL. 5, pp. 181-204, 1993.

[13] Roger M Orgando, Stephen S. Yau, Sying S. Liu, Norman Wilde, "*An Object Finder for Program Structure Understanding in Software Maintenance*", Software Maintenance: Research and Practice, VOL. 6, pp. 261-283, 1994.

[14] Ralph E. Johnson, "*Documenting Frameworks using Patterns*", OOPSLA '92 Proceedings, SIGPLAN Notices, 27(10): pp. 63-76, Vancouver BC, October 1992.

[15] Kent Beck, Ralph Johnson, "*Patterns Generate Architecture*", In European Conference on Object - Oriented Programming (ECOOP), ALNCS 821, Springer-Verlag, Bologna, Italy, pp. 320 - 343, July 1994.

[16] Henry C. Lucas, Donald J. Berndt, Greg Truman, " *A Reengineering Framework for Evaluating a Financial Imaging System*", Communications of the ACM, Volume 39, No. 5, pp. 86-96, May 1996.

[17] Kenneth C. Louden, "*Programming Languages Principles and Practice*", PWS Publishing Company, 1993.

[18] David Flanagan, "*Java in a nutshell*", O'Reilly & Associates Inc., 1996

[19] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensom. "*Object - Oriented Modeling and Design*". Prentice Hall, Englewood Cliffs, NJ, 1991.

[20] Erich Gamma, Richard Helm, Ralph Johnson. "*A catalog of object - oriented design patterns*". Technical Report in preparation, IBM Research Division, 1992.

[21] Christopher Alexander. "*The Timeless Way of Building*", Oxford University Press, New York, 1979.

[22] Yih-Farn Chen, Emden R. Gansner, Eleftherios Koutsofios, "*A C++ Data Model Supporting Reachability Analysis and Dead Code Detection*", Proc. ESEC/FSE 97, to appear.

[23] Bjarne Stroustrup, "*The C++ programming language*", Addison-Wesley Publishing Company, 1994.

VITA

Surname: Stier Given Names: Jochen

Place of Birth: Miltenberg, Germany

Education Institutions Attended:

University of Victoria 1991 to 1997

Degrees Awarded:

B.Sc. (Honours 1st class) University of Victoria 1994

Honours and Awards:

University of Victoria Fellowship 1996 to 1997

Publications:

PARTIAL COPYRIGHT LICENCE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library from any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

An Automated Approach to Object Oriented Design Pattern Detection and Extraction.

Author 

Jochen Stier
May 20, 1997