CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

This chapter presents the history, definitions and overviews of design pattern in general, and the structural design patterns in specific. It is important to highlight the structural design patterns in this chapter in order to understand how those patterns can be detected. In addition, this chapter discusses some design pattern detection and visualization systems and techniques. Moreover, this chapter provides examples of some design. Finally, this chapter discusses the reflection technique to extract the information from the code for the purpose of design patterns detection.

Design patterns are a valuable tool in software development. That shift of focus to a higher level of abstraction provides a common vocabulary when developers discuss about design. Design patterns allow developers to think of their designs at higher levels of abstraction; for example, instead of focusing on low-level details, such as how to use inheritance, complex systems can be thought as a collection of design patterns that already make the best use of inheritance.

2.2 History of design patterns

According to Stelting and Maassen (2002), Christopher Alexander, a professor of architecture at U.C. Berkeley, is the inspiration for design patterns in software development. In the late'70s, he published several books that introduced the concept of patterns and provided a catalogue of patterns for architectural design. In the next decade, a number of pioneers had developed patterns for software design. Beck and Cunningham

(1987) were among the first, discussing a set of Smalltalk design patterns in a presentation at the 1987 Object-Oriented Programming Systems, Languages, and Application conference (OOPSLA). James Coplien was another who wrote a book about C++ idioms, or patterns for C++ development, in the early '90s. Another important forum for the evolution of the patterns movement was Hillside Group, established by Kent Beck and Grady Booch.

According to Stelting and Maassen (2002), the best-known contribution to the popularity of design patterns was the book published by Gamma *et al.* (1995). The book introduced a comprehensive pattern language, and gave C++ examples for the patterns discussed. Another important book that gave popularity to patterns was the book Pattern-Oriented Software Architecture, A System of Patterns, by Buschamann, Meunier, Rohnert, Sommerlad and Stal (1996). After the publication of these two books, software community has taken more interest in design patterns. Java developers had taken an interest in applying design patterns in their projects. Design patterns in Java have become popular in presentations at conferences like JavaOne, as well as patterns columns in the Java trade journals.

2.3 Structural Design patterns

The scope of this study is on structural design patterns. Thus only these patterns are reviewed here. Stelting and Maassen (2002) stated that structural design patterns describe effective ways both to partition and to combine the elements of an application. There are seven structural patterns, which are Adapter, Bridge, Composite, Decorator, Façade, Flyweight and Proxy.

2.3.1 Adapter

According to Stelting and Maassen (2002), adapter acts as an intermediary between two classes, converting the interface of one class so that it can be used with the other. Implementing the Adapter pattern requires the following:

a. Framework – The Framework uses the Adapter. It constructs the ConcreteAdapter.
b. Adapter – The interface that defines the methods the Framework uses.
c. ConcreteAdapter – An implementation of the Adapter interface. It keeps a reference to the Adaptee and translates the method calls from the Framework into method calls on the Adaptee. This translation also possibly involves wrapping or modifying parameters and return types.
d. Adaptee - The interface that defines the methods of the type that will be adapted. This interface allows the specific Adaptee to be dynamically loaded at runtime.
e. ConcreAdaptee – An implementation of the Adaptee interface. The class that needs to be adapted so that the Framework can use this class.

Diagram of Adapter pattern is shown in Chapter 4.

2.3.2 Bridge

Bridge divides a complex component into separate but related inheritance hierarchies: the functional abstraction and the internal implementation (Stelting and Maassen, 2002). This makes it easier to change either aspect of the component. Implementing the Bridge pattern requires the following classes:

a. Abstraction – The Abstraction class defines the functional abstraction for the Bridge, providing standard behaviour and structure. It contains a reference to an Implementation instance. This Implementation instance is usually set with either a setter method (to allow modification at run-time) or through the constructor.

b. RefineAbstraction – The RefineAbstraction class extends the Abstraction class and provides additional or modified behaviour.

c. Implementation – The Implementation interface represents the under-lying functionality used by the Abstraction instances.

d. ConcreteImplementation – ConcreteImplementation implements the Implementation interface. It provides the behaviour and structure for Implementation classes.

Diagram of Bridge pattern is shown in Chapter 4.

2.3.3 Composite

Composite develops a flexible way to create hierarchies tree structures of arbitrary complexity, while enabling every element in the structure to operate with a uniform interface (Stelting and Maassen, 2002). The Composite has three elements:

a. Component – The Component interface defines methods available for all parts of the tree structure. Component may be implemented as abstract class when standard behaviour needs to be provided to all of the sub-types. Normally, the component is not instantiable; its subclasses or implementing classes, also called nodes, are instantiable and are used to create a collection or tree structure.

b. Composite – This class is defined by the components it contains; it is composed by its components. The composite supports a dynamic group of Components so it has methods to add and remove Component instances from its collection. The methods defined in the Component are implemented to execute the behaviour specific for this type of Composite and to call the same method on each of its nodes. These Composite classes are also called branch or container classes.

c. Leaf – The class that implements the Component interface and that provides an implementation for each of the Component's methods. The distinction between a Leaf class and a Composite class is that the Leaf contains no references to other Components. The Leaf classes represent the lowest levels of the containment structure.

Diagram of Composite pattern is shown in Chapter 4.

2.3.4 Decorator

Decorator provides a way to flexibly add or remove component functionality without changing its external appearance or function (Stelting and Maassen, 2002). For the Decorator pattern, implement the following:

a. Component - Represents the component containing generic behaviour. It can be an abstract class or an interface.

b. Decorator – Decorator defines the standard behaviours expected of all Decorators. Decorator can be an abstract class or an interface. The Decorator provides support for containment; that is, it holds a reference to a Component, which can be a ConcreteComponent or another Decorator. By defining the Decorator class hierarchy as a subclass of the component (s) they extend, the same reference can be used for either purpose.

c. One or more ConcreteDecorators – Each Decorator subclass needs to support chaining (reference to a component, plus the ability to add and remove that reference). Beyond the base requirement, each Decorator can define additional methods and/or variables to extend the component.

Diagram of Decorator pattern is shown in Chapter 4.

2.3.5 Façade

Facade provides a simplified interface to a group of subsystems or a complex subsystem (Stelting and Maassen, 2002). The following needs to be implemented for Façade:

a. Facade - The Class for clients to use. It knows the subsystems it uses and their respective responsibilities. Normally all client requests will be delegated to the appropriate subsystems.
b. Subsystem - This is a set of classes. They can be used by clients directly or will do work assigned to them by the Façade. It does not have knowledge of the Façade; for the subsystem the Façade will be just another client.

Diagram of Facade pattern is shown in Chapter 4.

2.3.6 Flyweight

Flyweight reduces the number of very-low detailed objects within a system by sharing objects (Stelting and Maassen, 2002).

To implement the Flyweight, the following is needed:

a. Flyweight - The interface defines the methods clients can use to pass external state into the flyweight objects.
b. ConcreteFlyweight – This implements the Flyweight interface, and implements the ability to store internal data. The internal data has to be representative for all the instances where you need the Flyweight.
c. FlyweightFactory - This factory is responsible for creating and managing the Flyweights. Providing access to Flyweight creation through the factory ensures

proper sharing. The factory can create all the flyweights at the start of the application, or wait until they are needed.

d. Client– The client is responsible for creating and providing the context for the flyweights. The only way to get a reference to a flyweight is through FlyweightFactory.

Diagram of Flyweight pattern is shown in Chapter 4.

2.3.7 Proxy

Proxy provides a representation of another object, for reasons such as access, speed, or security (Stelting and Maassen, 2002). For Proxy, implement the following:

a. Service – The interface that both the proxy and the real object will implement.

b. ServiceProxy – ServiceProxy implements Service and forwards method calls to the real object (ServiceImpl) when appropriate.

c. ServiceImpl - The real, full implementation of the interface. This object will be represented by the Proxy object.

Diagram of Proxy pattern is shown in Chapter 4.

2.4 Object-Oriented features for patterns realization and detection

Patterns are built on the theory and concepts of object-oriented programming, as surely as object-oriented approaches are built on procedural theory. Bansiya (1998) defined the key features of object orientation that is necessary for the realization and detection of design patterns. The features are inheritance, aggregation, uses, interfaces and polymorphism.

a) **Inheritance** is easy to identify by identifying base classes and subclasses.

b) **Aggregation** can have several forms in implementation. Aggregation could be implemented either as a physical containment, in which case the component objects are completely encapsulated inside the aggregate object, or by references of the component type that are initialised during program run time. Aggregation of the latter form provides the greatest flexibility, since the references can be set and changed at run time to refer to a variety of related objects. Typically, the aggregation relationship is detected by looking for data-member declarations in classes that are of other user-defined class types.

c) **Uses** relationship represents an arbitrary dependency between objects of two classes. This relationship is generally implemented through method parameters, and therefore detected by examining method parameter types.

d) **Interfaces** provide for a useful design and implementation technique, frequently referred to as "programming to an interface," which decouples users of services (clients) from implementation classes. It can be detected by identifying the implementation relationship between classes and interfaces.

e) **Polymorphism** allows parent-class-method overriding in child classes, objects of the child classes can be used and manipulated as objects of parent-class types. This concept offers great flexibility for programs to choose and select, at run time, the types of objects that are created and manipulated. Since the real value of polymorphism can only be seen when the program is executed, it is difficult to use polymorphism effectively in pattern detection and identification from static program implementations. However, polymorphism methods provide valuable information that can be used with pattern heuristics. In particular, the usage patterns of a class's

virtual method need to be tracked by other methods and instances in subclasses where the virtual methods are overridden.

2.5 Examples of Design Patterns

Examples of the use of design patterns can be found in many sources Geary (2001), Gamma *et al.* (1995) etc. In this section only examples of Composite and Decorator design patterns will be discussed to illustrate the usage of design patterns.

2.5.1 The composite pattern

As shown in Figure 2.1, the basic structure of a Composite pattern requires an aggregation relationship between parent and child classes. This relationship is typically implemented as a reference from the composite child class to a parent class and has a cardinality of 1-to-*N*. In terms of structure, there is a loop back from a child class to a parent class. Therefore, the detection of a composite pattern depends on the detection of this loop in branches of class hierarchies (Bansiya, 1998).

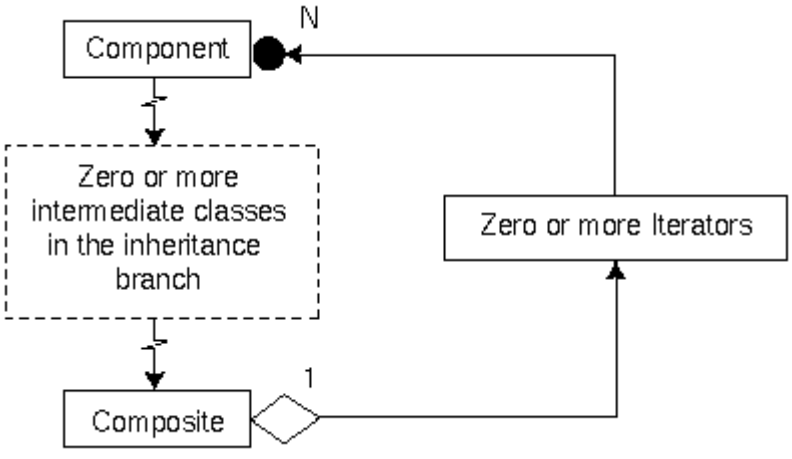

Figure 2.1 Composite pattern diagram (Bansiya, 1998)

According to Geary (2001), the Java's Abstract Window Toolkit (AWT) implements components and containers. Components can be added to a container, to facilitate complex user interface screens, user interface toolkits must allow nested containers, effectively composing components and containers into a tree structure. Moreover, it is important for components and containers in that tree structure to be treated equally, without having to distinguish between them. For example, when the AWT determines the preferred size of a complex layout containing nested components and containers, it walks the tree structure and asks each component and container for its preferred size. If that traversal of the tree structure required distinction between components and containers, it would unnecessarily complicate that code, making it harder to understand and modify.

The Composite pattern is used in AWT. The Composite pattern influences that containers *are* components, typically with an abstract class that represents both. In the AWT, that abstract class is *java.awt.Component*, the *superclass* of *java.awt.Container*; therefore, an AWT container can be passed to the *add(Component)* method from *java.awt.Container* because containers are components as shown in Figure 2.2.

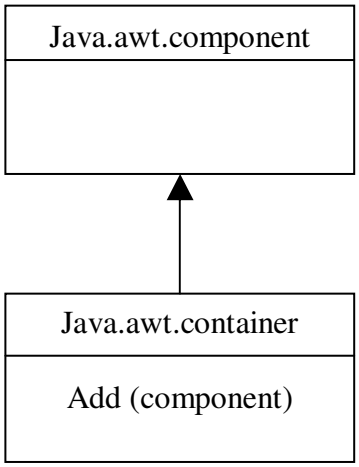

Figure 2.2 Example of composite design pattern: relationship between container and component

Figure 2.3 shows an applet that, by nesting containers, takes advantage of the AWT's Composite pattern implementation.

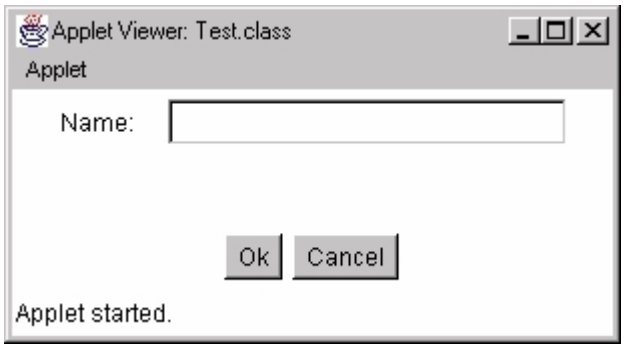

Figure 2.3 Use of the Composite pattern to nest AWT containers (Geary, 2001)

The code for the applet shown in Figure 2.3 is listed in Figure 2.4. The code created a panel that is an instance of WorkPanel that contains two other panels, as shown in Figure 2.5. Because the AWT implements components and containers using the Composite pattern, Java applets and applications, like the one listed in Figure 2.3, can easily nest components and containers in a tree structure. Additionally, components and containers can be treated equally.

```
import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Button;
import java.awt.Panel;
import java.awt.Label;
import java.awt.TextField;
public class Test extends Applet {
   public void init() {
      Panel        center = new Panel();
      WorkPanel workPanel = new WorkPanel(center);
      workPanel.addButton("Ok");
      workPanel.addButton("Cancel");
      center.add(new Label("Name:"));
      center.add(new TextField(25));
      setLayout(new BorderLayout());
      add(workPanel);
   }
}
class WorkPanel extends Panel {
   private Panel buttonPanel = new Panel();
   public WorkPanel(Panel centerPanel) {
      setLayout(new BorderLayout());
      add(centerPanel, "Center");
      add(buttonPanel, "South");
   }
   public void addButton(String label) {
      buttonPanel.add(new Button(label));
   }
}
```

Figure 2.4 Use of the composite pattern (Geary, 2001)

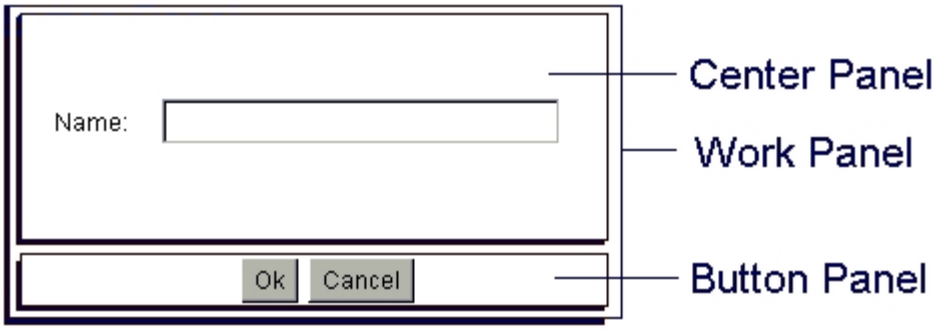

Figure 2.5 Nested AWT containers (Geary, 2001)

2.5.2 The decorator pattern

The basic structure of Decorator pattern is similar to the Composite pattern. It provides the ability to dynamically add responsibilities to objects. The key to implementing this pattern is also an aggregation relationship from a child class to a parent class. However, it differs from the Composite pattern in the cardinality of the aggregation relationship from the child class to parent class, which is 1-to-1 for the Decorator pattern. Both Composite and Decorator patterns require a recursive structure that can be extended in an open-ended fashion (Bansiya, 1998).

According to Geary (2001), the *java.io* package provides, among other things, a set of classes for reading input streams. Known as readers, each of those classes has a name that follows the pattern: *xxxReader*. Readers provide specialized functionality; for example, one reader reads from a file, another tracks line numbers, and yet another pushes characters back on the input stream. There are eight different readers exist to read input streams.

It is often necessary to combine the capabilities offered by *java.io* readers; for example, the programmer might want to read from a file, keep track of line numbers, and push certain characters back on the input stream, all at the same time. The java.io package's designers could have used inheritance to provide a wide array of such reader combinations; for

example, a *FileReader* class could have a *LineNumberFileReader* subclass, which could in turn have a *PushBackLineNumberFileReader* subclass. But using inheritance to compose the most widely used combinations of reader functionality would result in a veritable explosion of classes. The *java.io* package's designers create a design that enables the reader functionality to be combined in any desired way with only 10 reader classes. They used the Decorator pattern. Instead of using inheritance to add functionality to *classes* at *compile time*, the Decorator pattern lets the programmer add functionality to individual *objects* at *runtime*. That is accomplished by enclosing an object in another object. The enclosing object forwards method calls to the enclosed object and typically adds some functionality of its own before or after forwarding. The enclosing object known as a decorator, conforms to the interface of the object it encloses, allowing the decorator to be used as though it were an instance of the object it encloses. For example, the code listed in Figure 2.6 uses a decorator to read and print the contents of a file. The code also prints line numbers and transforms "Tab" characters to three spaces (Geary, 2001).

The *LineNumberReader* decorator encloses another reader; in this case, the enclosed reader is an instance of FileReader. The line number reader forwards method calls, such as read(), to its enclosed reader and tracks line numbers, which can be accessed with *LineNumberReader.getLineNumber*(). Because *LineNumberReader* is a decorator, the programmer can easily track line numbers for any type of reader.

```
import java.io.FileReader;
import java.io.LineNumberReader;

public class Test {
   public static void main(String args[]) {
      if(args.length < 1) {
         System.err.println("Usage: " + "java Test filename");
         System.exit(1);
      }
      new Test(args[0]);
   }
   public Test(String filename) {
      try {
         FileReader       frdr = new FileReader(filename);
         LineNumberReader lrdr = new LineNumberReader(frdr);

         for(String line; (line = lrdr.readLine()) != null;) {
            System.out.print(lrdr.getLineNumber() + ":\t");
            printLine(line);
         }
      }
      catch(java.io.FileNotFoundException fnfx) {
         fnfx.printStackTrace();
      }
      catch(java.io.IOException iox) {
         iox.printStackTrace();
      }
   }
   private void printLine(String s) {
      for(int c, i=0; i < s.length(); ++i) {
         c = s.charAt(i);

         if(c == '\t') System.out.print("   ");
         else          System.out.print((char)c);
      }
      System.out.println();
   }
}
```

Figure 2.6 Use the Decorator pattern (Geary, 2001)

2.6 Existing Detection systems and techniques

Patterns detection is the identification and extraction of patterns from the code. Section 2.6.1 describes some of the existing pattern detection systems. Some detection techniques will be discussed in section 2.6.2. Section 2.6.3 discusses and compares among the existing detection systems and techniques.

2.6.1 Existing Detection systems

2.6.1.1 Pat system

The Pat system is a design recovery tool for C++ applications (Kramer and Prechelt, 1996). It extracts design information directly from C++ header files and stores it in a repository. The fundamental idea for the automated search in Pat is to represent both patterns and designs in PROLOG and let the PROLOG engine do the actual search. The basic design information itself is extracted from source code by the structural analysis mechanism of a commercial object-oriented CASE tool. Patterns are defined as PROLOG rules and the design information is translated into facts. The Pat system can only be used to detect structural design patterns. Both patterns and examined designs are represented in PROLOG format. The actual matching work is done by a PROLOG engine. Pat does not intend to do very detailed program understanding or design recovery. Its limitation is that it has difficulty in dealing with behavioral patterns, since a lot of semantic information is required.

Figure 2.7 shows the architecture of the Pat system. The system follows the following steps to detect the patterns:

1-The structural analysis mechanism of the CASE tool is used to extract design information from C++ header files and represent it in the repository in Object Modelling Technique (OMT) form. The resulting part of the repository is called *D*

2-Another program prolog converts *D* into PROLOG representation;

3-A PROLOG query *Q* detects all instances of design patterns from *the repository* in the examined design *D*.

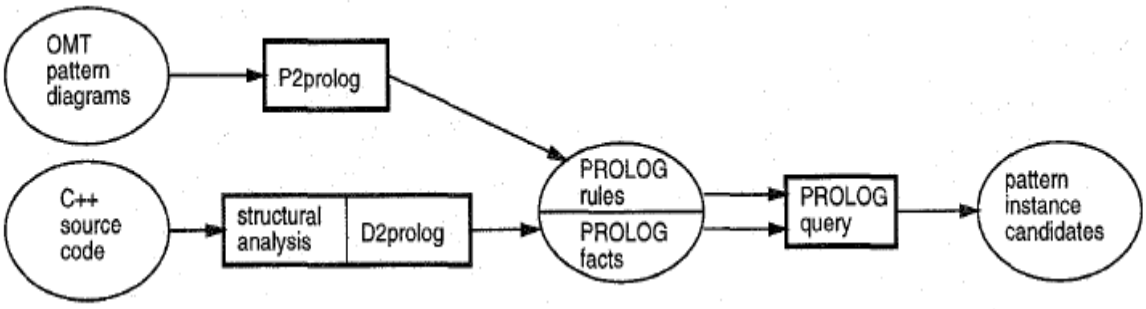

Figure 2.7 Architecture of the Pat System (Kramer and Prechelt, 1996)

2.6.1.2 JBOORET (Jade Bird Object-Oriented Reverse Engineering Tool)

JBOORET adopts a parser-based approach to extract the higher-level design information and conceptual model from system artifacts (Mei *et al.*, 2001). As shown in Figure 2.8, the JBOORET for C++ consists of three major components: a data extractor, a knowledge manager and an information presenter. Separating data extraction from information representation, this architecture avoids repeating analysis process for each higher-level model extraction. As the front-end, the data extractor is the only part that processes language dependent data. This design enables JBOORET to be easily adapted into reverse engineering a system written in a language other than C++. The knowledge manager contains conceptual models of the developing language. The information presenter component organizes and generates design information according to the user's preference.

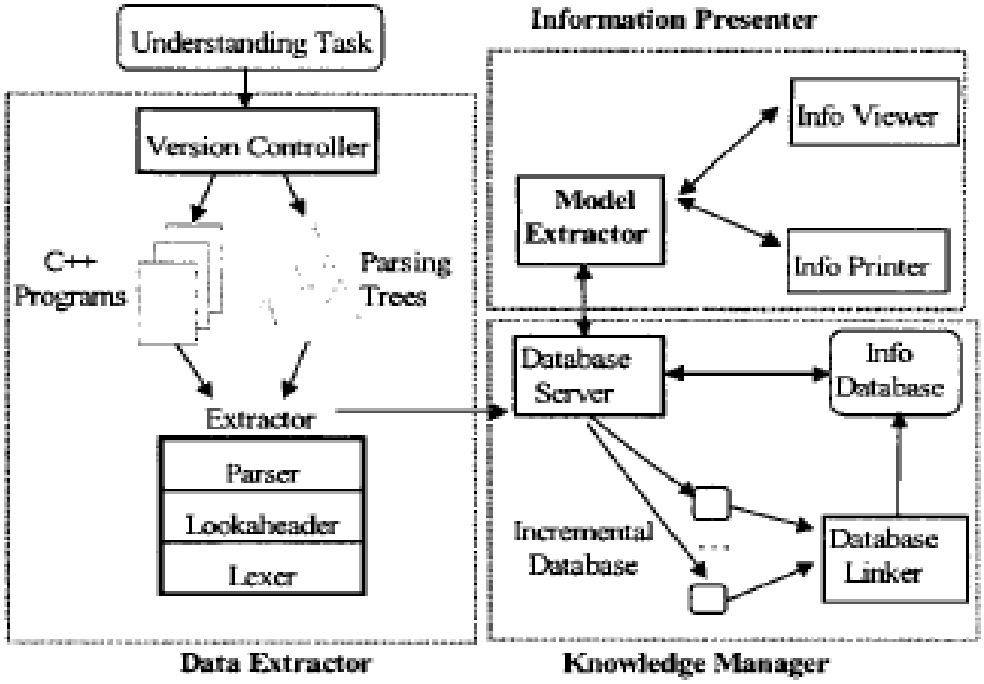

Figure 2.8 JBOORET Architecture (Mei et al., 2001)

2.6.1.3 DPVK (Design Pattern Verification toolkit)

Wang and Tzerpos (2005) claim that DPVK is able to detect several different design patterns by examining both the static structure and the dynamic behavior of a system written in Eiffel. They presented the first design pattern detection tool for systems written in Eiffel. Eiffel is a pure object oriented language with many distinct features that require a different approach to pattern detection, such as generic and executable contracts.

DPVK uses two definitions to identify instances of design patterns, as well as to differentiate between different design patterns. One definition is based on the static structure of the pattern and the other is based on its dynamic behavior. As shown in Figure 2.9, DPVK includes four stages: static fact extraction, candidate instance discovery, false positive elimination, and manual evaluation. The first three stages directly correspond to modules in DPVK implementation.

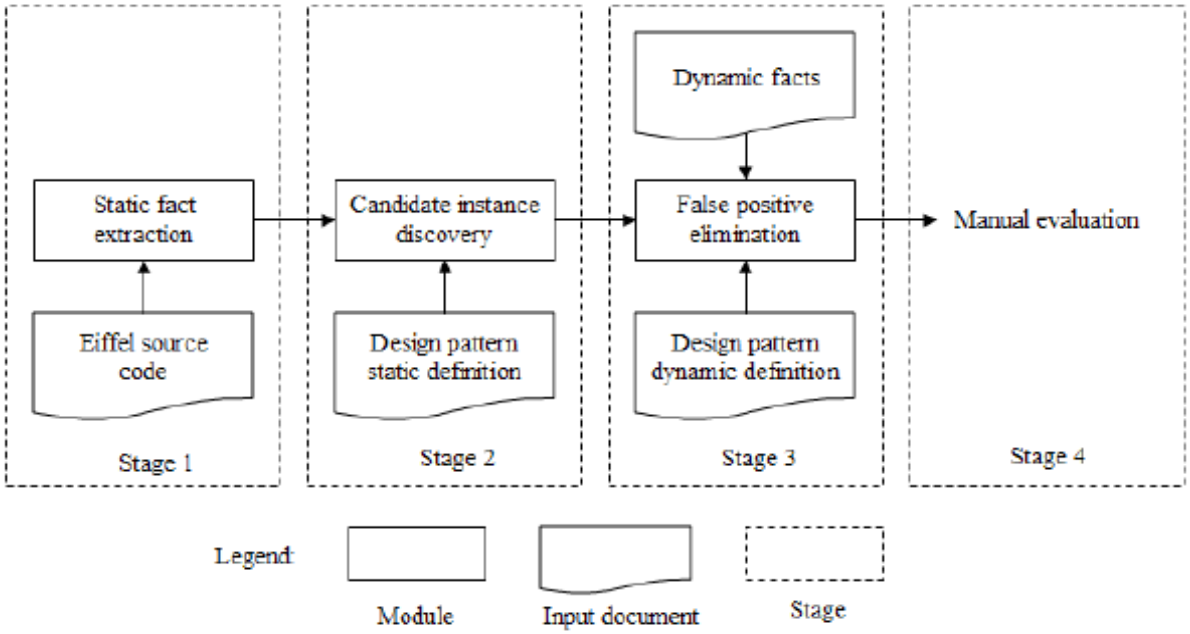

Figure 2.9: The structure of DPVK (Wang and Tzerpos, 2005)

DPVK is quite effective in detecting design pattern instances while eliminating false positive. DPVK does not take certain factors into consideration, such as the signature of methods, the class type (concrete/abstract), language keywords etc. For example, the ONCE keyword in Eiffel helps developers implement the Singleton pattern. Therefore, its identification is essential to the detection of the Singleton pattern. However, DPVK does not recognize the ONCE keyword yet. Finally, the current user interface for DPVK is textual. A graphical user interface that can depict discovered instances in Unified Modelling Language (UML) or Business Object Notation (BON) would be helpful for the manual evaluation phase.

2.6.1.4 SPQR: Flexible Automated Design Pattern Extraction from Source Code

Smith and Stotts (2003) have suggested a method to extract the design patterns called System for Pattern Query and Recognition (SPQR). SPQR does not seek statically to

encode each variant of the patterns that the user wishes to find. SPQR system finds pattern variants that were not explicitly defined, but instead are inferred dynamically during code analysis by a theorem prover. Smith and Stotts (2003) have also used a logical inference system to reveal large numbers of patterns and their variations from a small number of definitions by encoding in a formal denotational semantics a small number of fundamental Object-Oriented concepts (elemental design patterns), encoded the rules by which these concepts are combined to form patterns (reliance operators), and encoded the structural/behavioral relationships among components of objects and classes (rho-calculus).

As shown in Figure 2.10, SPQR is a tool that performs the analysis from source code and produces a final report. Source code is first analyzed for particular syntactic constructs that correspond to the ρ-calculus concepts. The gcc has the ability to emit an abstract syntax tree suitable for such analysis. The first tool, gcctree2oml, reads this tree file and produces an XML description of the object structure features. An intermediary step was chosen so that various back ends could be used to input source semantics to SPQR. A second tool, oml2otter then reads this Object XML file and produces a feature-rule input file to the automated theorem prover; in the current package OTTER are being used. OTTER is a fourth-generation Argonne National Laboratory deduction system that searches for instances of design patterns by inference based on the rules. Finally, proof2pattern analyzes the OTTER proof output and produces an Object XML pattern description report that can be used for further analysis, such as the production of UML diagrams.

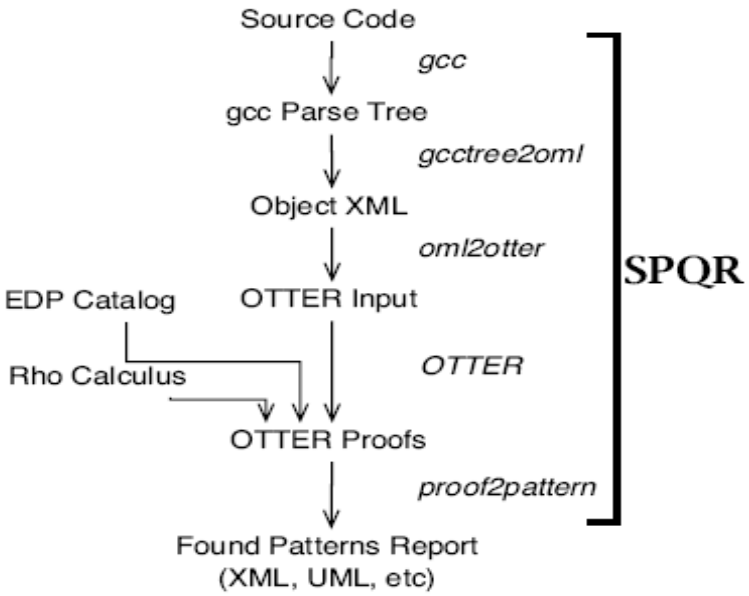

Figure 2.10: SPQR Outline (Smith and Stotts, 2003)

SPQR can allow new design patterns to be added to the catalogue used for query and can be adapted to work at multiple levels of formal analysis, depending on the particular need. However, SPQR cannot detect the design pattern directly. SPQR must transform the code into XML then it can do some analysis and after that patterns can be detected.

2.6.2 Existing Detection techniques

2.6.2.1 Bit-vector algorithm

Kaczor *et al.* (2006) proposes an efficient approach of design pattern identification using a high-performance bit-vector algorithm. Bit-vector algorithms are widely used in pattern matching and in bio-informatics. A bit-vector algorithm was used to perform the structural matching between micro architectures and design motifs.

First, models of the design motif of the program are converted into digraphs. Then, these digraphs are converted into Eulerian digraphs to generate unique string representations of the design motif and of the program. Finally, a new bit-vector algorithm

is applied on the string representations to identify exact and approximate occurrences of the design motif in the program efficiently.

The algorithm is interesting because of its efficiency. However, the generation of the string representations requires choosing a root vertex for the traversal of the digraph. The choice of the root vertex impacts the string representations.

2.6.2.2 Negative search criteria

Streitferdt *et al.* (2005) proposed an algorithm which focuses on negative search criteria for patterns. In this algorithm, the structure of each pattern is defined by its required, forbidden (negative) and uncertain elements. All required elements have to be present, forbidden elements are not allowed in the structure and uncertain elements can be present, but will be ignored in order to come to a successful identification of a pattern by the algorithm.

The key problem for the algorithm is the lack of a good detection system, a system that can address the implementation variants for a given pattern. So this algorithm is considered not accurate enough to detect patterns.

2.6.2.3 Multi-stage reduction strategy using software metrics

Antoniol *et al.* (2006) presents a conservative approach, based on a multi-stage reduction strategy using object-oriented software metrics and structural properties to extract structural design patterns from object-oriented design or code. Code and design are mapped into an intermediate representation, called Abstract Object Language (AOL). Software metrics, as well as structural properties, are extracted from an Abstract Syntax Tree (AST) produced by parsing the AOL representation. Pattern structure is then exploited to further reduce the search space. A pattern can be conceived as a graph in which nodes are classes

and edges correspond to relations. Such representation contains information about classes, their methods and attributes as well as relations among classes. Object-oriented software metrics are used to determine pattern candidate sets. If a pattern is in the code it is reported in the results. Sometimes, extra patterns are reported: to further reduce false detection, design pattern constraints in term of method delegation have been exploited. Method delegation means that a class implements one operation by simply calling an operation of another class to which it is associated. Figure 2.11 shows a screen-shot of the system interface.

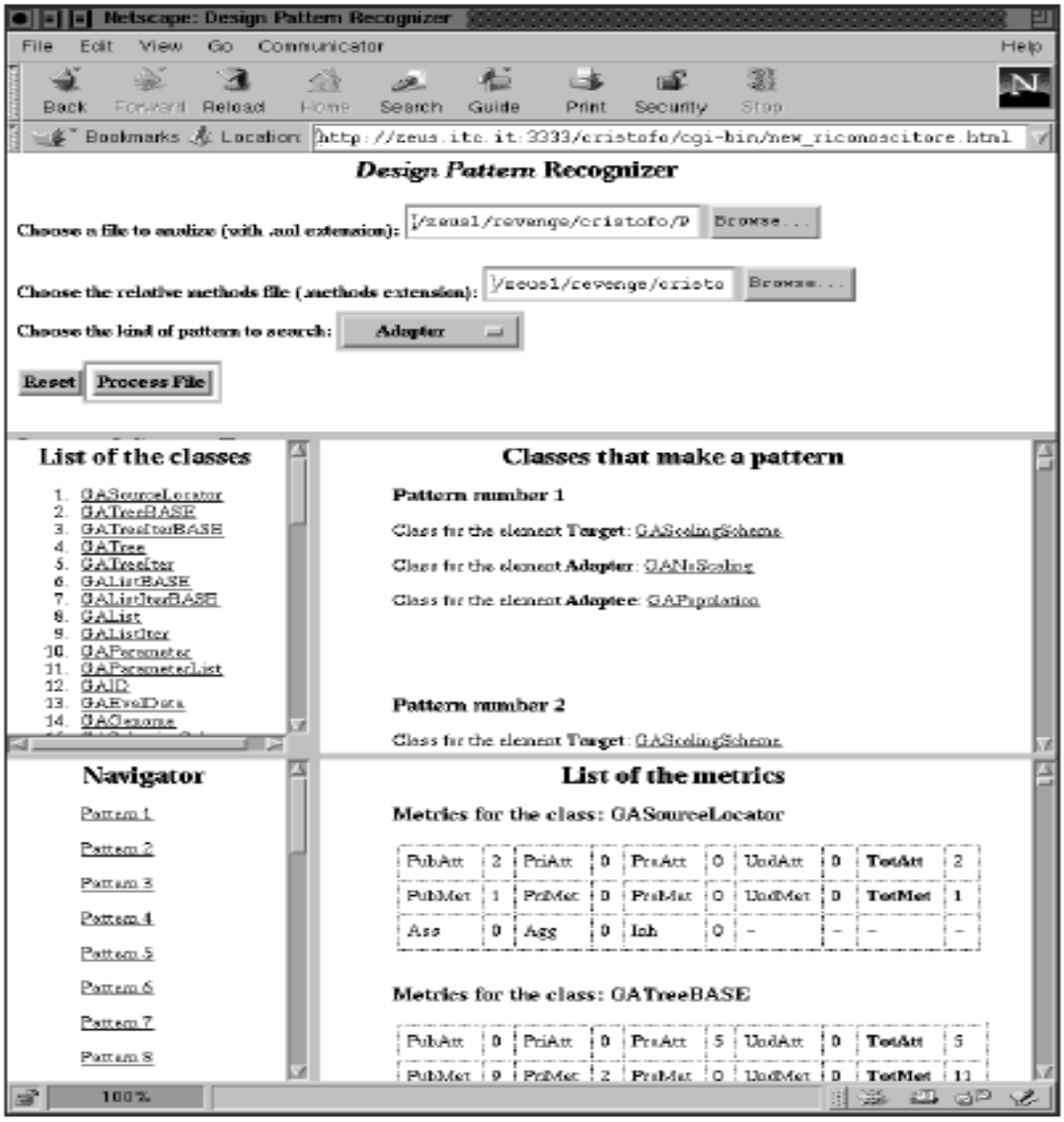

Figure 2.11 User interface screen-shot of Antoniol's system (Antoniol, 2006)

The strategy process consists of the following activities:

a) **AOL Representation Extraction:** in this phase an AOL representation is recovered from both design and code.

b) **AOL Representation Parsing:** an AOL Parser builds an AST, which subsequent phases rely on.

c) **Class Metrics Extraction:** a Metrics Extractor computes metrics on the class declarations contained in the AOL AST and decorating the AST with such information.

d) **Pattern Recognition:** a multi-stage Constraint Evaluator process localizes design pattern instances.

e) **GUI:** the AOL input file selection, the recognition process, extracted metrics and pattern results visualization are implemented using Web technology and HTML pages displayed by Web browsers.

Antoniol *et al.* (2006) focused recovery effort on five of the seven structural design patterns: the adapter, the proxy, the composite, the bridge and the decorator. The author developed a distributed system based on Java and WWW technology to automate the design pattern recovery process. The multi-stage search space reduction approach maintains the response time at an acceptable level with an average precision of 55%. Time execution and memory requirements result scarified in favor of portability

2.6.2.4 Static and Dynamic Analysis combined

Heuzeroth *et al.* (2003) presented approach to detect design patterns by combining static and dynamic analyses. Therefore, the program is executed under investigation and the

executions of the candidate instances found by the static analysis are monitored with respect to the dynamic restrictions. Figure 2.12 illustrates this approach.

Figure 2.12: Static and Dynamic Analyses of Patterns (Heuzeroth et al., 2003)

The static analysis computes potential program parts playing a certain role in a design pattern. The dynamic analysis further examines those candidates. Static and dynamic analyses can be considered as filters that narrow the set of candidates in two steps. The results of dynamic analyses depend on an execution of the candidate instances. Methods not executed at run time cannot be evaluated with respect to the dynamic pattern thus providing no information.

The static analyses are done with Recoder that was developed by Ludwig *et al.* (2001), a tool to read, analyse and manipulate Java programs. In principle, Recoder is a compiler front end. It reads the source code and constructs the abstract syntax tree (AST). Then it performs the static semantic analyses where it resolves names and types.

Recoder provides all these entities and relations as an API. In order to analyze the structure of a program, a program is written to access this interface (API). The elements of candidates of pattern instances retrieved by the static analysis are nodes in the AST accessible via Recoder. Examples of such elements are classes and methods that potentially play different roles in a pattern.

The analysis tool did not detect any false positive patterns (in case a pattern was found but is not implemented in the source code) and did detect all real patterns. The detection of false positives was excluded by screening the analysis results. An estimation of true negatives (in case a pattern was not found but is implemented in the source code) was not possible. However, there is a clear indication that the dynamic analyses should be improved. According to Gutzmann (2008), Recorder is already useful but it also still needs polishing since it has some bugs.

2.6.3 Comparison among the existing detection systems and techniques

Table 2.1 shows a comparison among the existing detection systems while Table 2.2 shows a comparison among the existing detection techniques that were mentioned in this literature.

Table 2.1

Comparison among the existing detection systems

Criteria	**PAT**	**JBOORET**	**DPVK**	**SPQR**
Design Patterns that can be Detected	Structural	Classes and relationships only	All	All
Programming language used to develop the system	PROLOG	C++	Eiffel language	Calculus
Samples Codes	Four C++ files (zApp,LEDA, NME,ACD)	76 sample C++ projects	Eiffel system	C++
Detection Method	It uses Paradigm Plus Case Tool to extract info about classes from C++ header files	Adopts a parser-based approach to extract the higher-level design	It can detect several different design patterns by examining both the static structure and the dynamic behavior of a	SPQR does not seek statically to encode each variant of the patterns that the user wishes to find. SPQR

		information and conceptual model from system artifacts	system	system finds pattern variants that were not explicitly defined, but instead are inferred dynamically during code analysis by a theorem prover.
Strengths	Fast and simple way to recover design information from source code.	-Its flexible user interface can assist users to browse the detailed information of design and source models -It conforms to the principle of separating data extraction from information presentation	It is quite effective in detecting design pattern instances while eliminating false positive	SPQR can allow new design patterns to be added to the catalogue used for query and can be adapted to work at multiple levels of formal analysis
Weaknesses	- Some information that are relevant for a precise search for pattern instances can not be extracted by Paradigm Plus - It does not read behaviour of methods. Composite Pattern difficult To detect.	It needs to be enhanced to detect design patterns and not only classes and relationships among them	- DPVK does not take certain factors into consideration, such as the signature of methods and the class type - current user interface for DPVK is textual.	SPQR cannot detect the design pattern directly. SPQR must transform the code into XML then it can do some analysis and after that patterns can be detected.
Precision rate	40 %	Unknown	>50 %	Unknown

Table 2.2

Comparison among the existing detection techniques

Criteria	**Bit-vector**	**Negative search**	**Multi-stage reduction**	**Static and Dynamic Analysis**
Design Patterns that can be detected	Abstract Factory and Composite	All	Adapter, proxy, composite, bridge, and decorator	Observer, Composite, Mediator, Chain of responsibility, Visitor
Technique	Software Metrics	Negative search criteria for patterns	Object-Oriented software metrics	Abstract syntax tree (AST)
Samples Codes	Java (JhotDraw, Juzzle, Quick UML)	Java (Drawlet, Awt, Tomcat)	C++ systems (galib, gruff, LEDA, libg++, mec, Socket)	Itself and Java SwingSet
Detection Method	A bit-vector algorithm was used to perform the structural matching between micro architectures and design motifs	The structure of each pattern is defined by its required, forbidden (negative) and uncertain elements. All required elements have to be present, forbidden elements are not allowed in the structure and uncertain elements can be present, but will be ignored in order to come to a successful identification of a pattern by	Code and design are mapped into an intermediate representation, called Abstract Object Language (AOL). Software metrics, as well as structural properties, are extracted from an Abstract Syntax Tree (AST) produced by parsing the AOL representation. Pattern structure is then exploited to further reduce the search space. A pattern can be conceived as a graph in which nodes are classes	The static analysis computes potential program parts playing a certain role in a design pattern. The dynamic analysis further examines those candidates. Static and dynamic analyses can be considered as filters that narrow the set of candidates

		the algorithm.	and edges correspond to relations.	
Strengths	High performance	Can detect all design patterns	-The approach maintains the response time at an acceptable level - Portability	Combined both static and dynamic analyses.
Weaknesses	Generation of the string representations requires choosing a root vertex for the traversal of the digraph. The choice of the root vertex impacts the string representations.	It does not address different implementation variants for a given design pattern	- The assumption that the micro-architecture accurately reflects the design motifs, which is rare. - The theoretical quantification of roles does not reduce the search space significantly	-No visualization -Static analysis done with Recorder that needs some polishing since it has some bugs.
Precision rate	Unknown	70 %	55 %	Unknown

2.7 Visualization techniques

The visualization techniques of design patterns are similar to the ones for code visualization. Visualizing code concentrates on viewing all the details about the code such as the classes, methods, attributes, and relationships among classes. Visualising design patterns is more specific and concentrates on the pattern instances such as participating classes, and relationships among those classes. In this section, some of the design patterns visualization techniques are discussed.

2.7.1 Programs visualization

Roman and Cox (1992) identified four axes along to classify program visualization systems. The first and the fourth relate to the domain and the range of the mapping, respectively as shown in Figure 2.13.

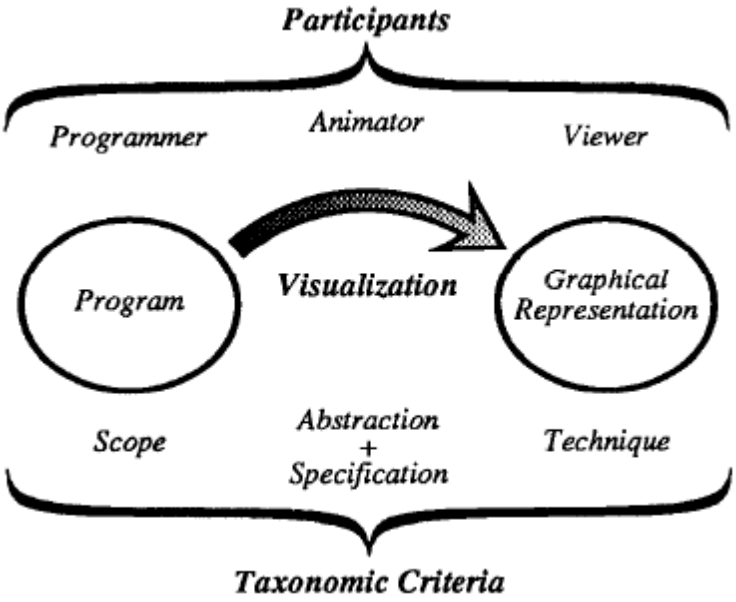

Figure 2.13: Four axes of program visualization (Roman and Cox, 1992)

The four axes are:

(1) Scope—what aspect of the program is visualized? The domain of an individual visualization is a program. Formally, a program can be characterized by its code, by its data and control states, and by its execution behaviour. Visualization systems often limit their scope to a subset of these program aspects.

(2) Abstraction—what kind of information is conveyed by the visualization? The issue is the level of abstraction associated with the concepts presented in a graphical form by the particular system. Highlighted code, for instance, offers a very low-level representation of the control state. An animation, on the other hand, augments a behaviour depiction with choreographed explanatory information.

(3) Specification Method—how is the visualization constructed? This issue is essential to understanding the power and flexibility of a particular system. Some systems provide "hardwired" mappings, while others may allow for arbitrary redefinitions of the mapping;

some focus on mapping program states while others focus on events; some require modification of the code while others do not.

(4) Technique—how is the graphical representation used to convey the information? This criterion considers issues having to do with effective visual communication. They include the visual vocabulary, the use of specific visual elements to convey particular kinds of information, the organization of visual information, and even the order in which material is presented to the viewer.

2.7.2 Visualizing design patterns using UML

Design patterns are usually modeled and documented in the Unified Modeling Language (UML). However, Dong *et al.* (2005) claim that the constructs provided by the standard UML are unable to visualize design patterns in their applications and compositions.

Dong *et al.* (2005) presented a web service (VisDP) for explicitly visualizing design patterns in UML diagrams. Figure 2.14 shows a diagram of the overall system architecture. This web service is developed based on a UML profile containing new stereotypes, tagged values and constraints for visualizing design patterns in UML diagrams. With this service, the user is able to identify design patterns by moving the mouse and viewing colour changes in UML diagrams. Additional pattern-related information can be dynamically displayed based on the mouse location. VisDP is deployed as a web service and can be accessed through the tool website and providing, as an input, a XML file generated by the plug-ins of a UML tool, such as Rational Rose as shown in Figure 2.15.

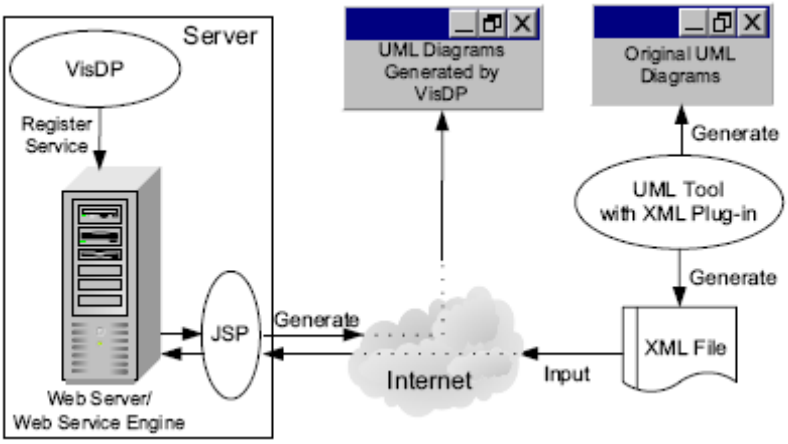

Figure 2.14: Overall System Architecture (Dong et al., 2005)

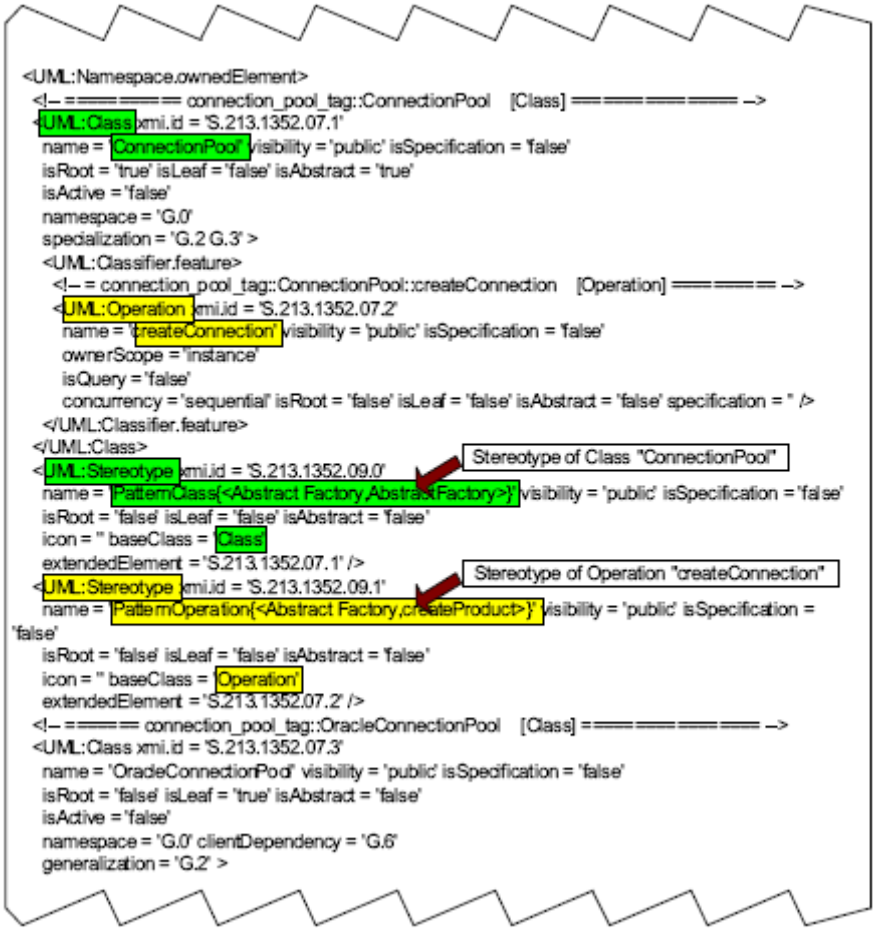

Figure 2.15: Input XML File for Class Diagram (Dong et al., 2005)

The VisDP can hide or show pattern-related information on demand. When the pattern-related information is hidden, the diagrams are just like the ordinary UML diagrams. When the pattern-related information is shown, the end user can identify the design patterns that a class (operation/Attribute) participates and the roles it plays by different colours and on-demand information shown in the diagram. The software designer can move his/her mouse onto the modelling element (e.g., class, operation, attribute) in question. All classes that participate in the same pattern as the class under the mouse are changed to the same colour. Figure 2.16 shows Connection Pool Class Diagram with the pattern information.

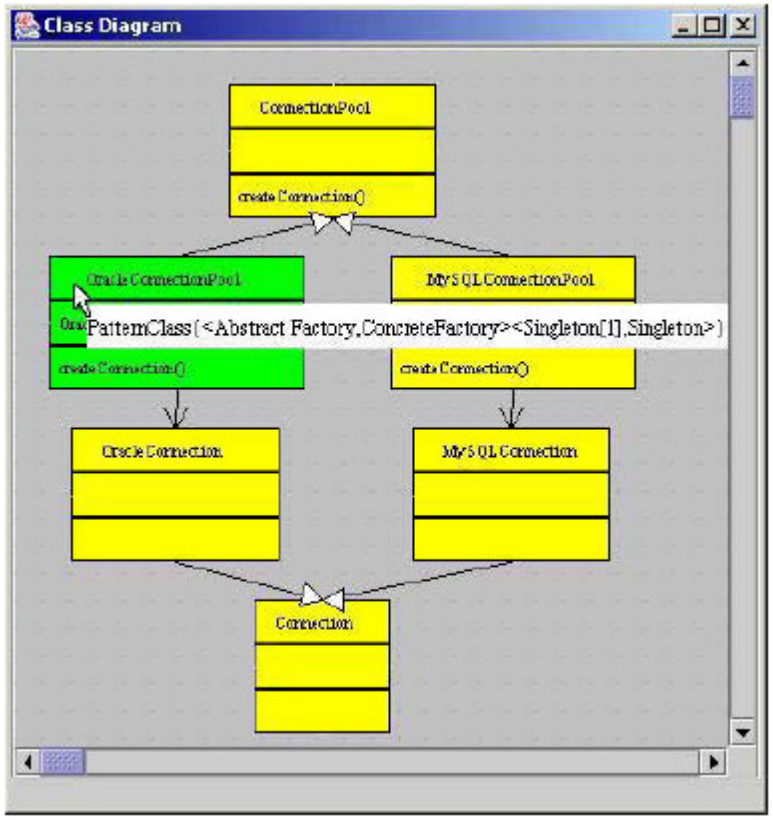

Figure 2.16

Connection Pool Class Diagram with the Pattern Information (Dong et al., 2005)

The VisDP is a useful tool that can aid in visualizing design patterns. However, since VisDP is a web service, a developer must have internet connection in order to use this web service. Moreover, the developer must have Java plug-in 1.3. VisDP can visualise design patterns from XML files only.

2.7.3 Visualizing design patterns using virtual reality and hypertext

Callaghan and Hirschmiiller (1998) describe a form of visualization that makes use of a combination of virtual reality and hypertext to represent the structure and behaviour of object-oriented design patterns and simple Java programs. An objective for the visualization was to explore the possibility of visualizing a part of the pattern or software fragment in virtual reality and complementing this view with hypertext. This aspect is intended to combine the benefits of both techniques.

Callaghan and Hirschmiiller (1998) have used traditional diagrams and joined them together in a higher dimensional visualization so that the traditional diagrams are then special cases of the resulting view. This metaphorical use of traditional diagrams maintains familiarity to aid learning and enhance understanding. Visualization in a virtual environment offers four dimensions (three spatial dimensions and time) in contrast to the two dimensions of paper-based diagrams. Callaghan and Hirschmiiller (1998) concentrated on the class, object and interaction diagrams**.** Figure 2.17 shows the combined view of all three kinds of diagrams, as represented in its current work.

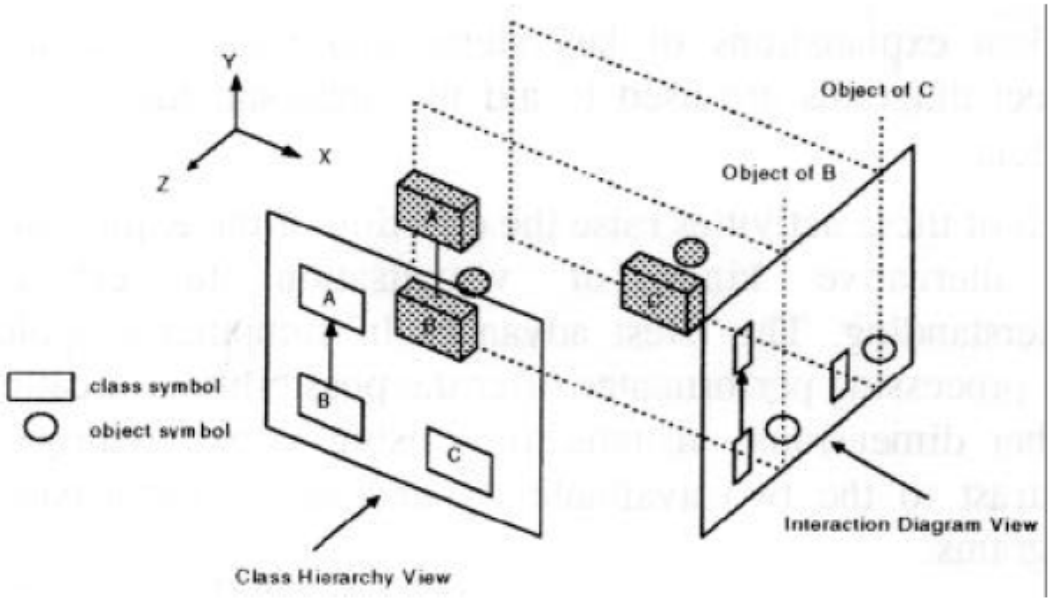

Figure 2.17: The three spatial dimensions (Callaghan and Hirschmiiller, 1998)

In this way, all three kinds of diagrams are visualized at once, while maintaining as much familiarity with the two-dimensional diagrams as possible. This approach is good because when representing method calls and pointers, both objects and their classes are shown in contrast to the traditional object or interaction diagrams. Moreover, basic descriptions of the classes are shown upon clicking on the classes in the visualization. However, this approach can slow down the system since it uses virtual reality. Thus high power computer is needed.

2.8 Reflection technique for design patterns detection

Reflection is a technique for extracting the relevant data from the code. It is important to highlight this technique and shows how it works since this technique is applicable in the extraction of the information from the code during the development of the tool proposed in this study.

2.8.1 Definition of Reflection

Olsen (2007) defines Reflection as a feature in the Java programming language. It allows users to look at a program structure by giving users a window into the fundamental features of the language.

Darwin (2007) highlighted that the Reflection API allows users to operate on classes and objects in ways such as the following:

1 Load a class file into memory at run time, knowing only its name.

2 Given a class, examine its methods, fields, constructors, annotations, and so on.

3 Given a class, invoke constructors and methods.

4 Given a class and an instance, access fields.

5 Given an interface, create proxies for it dynamically.

2.8.2 Mechanism of Reflection

Mcmanis (1997) explained that the mechanism used to fetch information about a class is the use of the class named "Class". The special class "Class" is the universal type for the meta information that describes objects within the Java system. Class loaders in the Java system return objects of type Class. The developer can use the "forName" method to load a class of a given name. After that class is loaded then the developer can access this loaded class to extract the required information. Here are some of the methods that can used to extract some information from the loaded class:

1 getConstructor, getConstructors, getDeclaredConstructor

2 getMethod, getMethods, getDeclaredMethods

3 getField, getFields, getDeclaredFields

4 getSuperclass

5 getInterfaces

6 getDeclaredClasses

2.8.3 Reflection Example

To see how reflection works, consider this simple example shown in Figure 2.18. In line 7, “forName” method is used to load “RefMethods” class. In line 8, “getDeclaredMethod” method is used to extract the declared methods in the class ”RefMethods”. In line 10, the names of extracted methods are printed.

As shown in the output, the full names of methods were printed along with the package name. Other information of an extracted method can be shown such as the specific method name, return type, return value, method parameters, and so on.

```
1  import java.lang.reflect.*;
2
3  public class RefMethods {
4      public RefMethods(){
5
6         try {
7            Class c = Class.forName(“RefMethods”);
8            Method m[] = c.getDeclaredMethods();
9           for (int i = 0; i < m.length; i++)
10            System.out.println(m[i].toString());
11         }
12         catch (Throwable e) {
13            System.err.println(e);
14         }
15      }
16      public int method1(){
17      int i=0;
18
19      return i;
20      }
21      public String method2(){
22      String s=”hi”;
23
24       return s;
25      }


   }
```

Figure 2.18: Reflection example

The output is:

public java.lang.String RefMethods.method1()

public java.lang.int RefMethods.method2()

2.9 Summary

This chapter has presented the background of the fundamental issues related to design pattern detection and visualization. This chapter presented the history, definitions and overviews of design pattern in general, and the structural design patterns in specific and provided examples of some design besides presenting some design pattern detection and visualization systems and techniques and reflection technique to extract the information from the code for the purpose of design patterns detection.