Abstract

Design patterns have been quickly adopted by the Object-Oriented community, in particular, since the publication of “Design Patterns: Elements of Reusable Object-Oriented Software” [1]. They offer highly reusable and high quality Object-Oriented design components, which increase the productivity and quality of software. However, it is users onus to select and to apply them into their design problems. Additionally, success in applying design patterns depends on the understanding and implementation of patterns. In order to assist developers, methods and tools for design patterns need to be developed. This paper describes an approach (meta-model of design patterns) that facilitates the formal representation of object oriented design patterns and automatic code generation for design patterns. In particular, the proposed approach generates abstract codes for patterns in various object oriented programming languages. The proposed method was applied to some of the patterns selected from the Gang-of-Four catalog, and the results are also presented.
1. INTRODUCTION

Object oriented design patterns capture knowledge and experience in building object-oriented software. A design pattern describes an abstract solution to a recurring design problem in a systematic and general way. But beyond a description of the problem and its solution, software developers need deeper understanding to tailor the solution to variant of the problem. Hence a design pattern also explains the applicability, trade-offs, and consequences of the solution [1].

Over the past couple of years, a vibrant research and user community has sprung up around this topic. Pattern-related discourse has flourished at object-oriented conferences, so much so that there is now a conference devoted entirely to patterns. Books [1, 4, 8] and at least one non-profit organization (The Hillside Group) have appeared to further the field. One of the most widely cited books is Design Patterns: Elements of Reusable Object-Oriented Software [1], a catalog of 23 design patterns culled from numerous object-oriented systems. Design Patterns has proven popular with novice and experienced object-oriented designers alike. It gives them a reference of proven design solutions along with guidance on how to implement them. The discussions of consequences and trade-offs furnish the depth of understanding designers need to customize the implementation to their situation. Additionally, the names of the patterns collectively form a vocabulary for design that helps designers to communicate and to share knowledge and experience.

Design patterns offer highly reusable and high quality Object-Oriented design components, which increase the productivity and quality of software. However, it is users onus to select and to apply them into their design problems. Additionally, success in applying design patterns depends on the understanding and implementation of patterns during the implementation process, and experienced programmers can quickly transform the specification into code since they have probably implemented the Design Patterns many times. For novice programmers, the process of coding the design pattern is more difficult and error-prone. This is partly due to the fact that design patterns are written documents that are subject to human interpretation. This makes them vulnerable to the ambiguities in natural language. An incorrect interpretation of a pattern can lead to an incorrect implementation. It would be beneficial to use CASE tool of design patterns that generate code. They reduce implementation time, are less prone to programmer error, promote rapid prototyping and code reuse, support performance tuning, and provide better overall software engineering benefits.

The application of a design pattern can be decomposed in three distinct activities:

1. The choice of the right pattern, which fulfils the users’ requirements
2. Its adaptation to these requirements (the term instantiation is commonly used to identify this task)
3. The production of the code required for its implementation.

Automating the last two tasks is a very challenging issue for the design patterns community, and it galvanizes a lot of research work. We put more emphasis in developing a new framework that assists users to solve these problems.
This paper proposes a new framework for presentation of object oriented design patterns. Application of the framework for automatic code generation is also presented.

The rest of the paper is organized as follows. Section 2 summarizes the related work. Our methodology for representation of patterns and its applicability is discussed in Section 3. Section 4 presents the results obtained. Finally the paper is concluded in Section 5.

2. RELATED WORK

There are several approaches for formal representation of design patterns; a lattice is defined as a set of qualifications on the structural, behavioral and relational aspects of programs. LePUS is a declarative specification language that uses higher order monadic logic. Classes and functions in the design pattern is represented by primitive variables. Predicates over these variables describe characteristics or relationships between the elements. A visual notation for LePUS formulas consisting of squares, ovals and triangles are also included. These icons represent variables or sets of variables and commented directed arcs representing the predicates. It is difficult for average software developers to work with LePUS because it is based on mathematics and formal logic. Tool support for LePUS is based on Prolog and lacks support for the visual notation. The LayOM is an extended object model; it defines new components such as layers, states and categories. The layers encapsulate the objects, so the messages sent or received by the object have to pass the layers. Each layer, when it gets a message, converts the message into a passive message object and evaluates the contents to decide the course of action. Previously, layers have been used to represent relations between objects. Relations in LayOM can perform all kinds of behavior, a response to a message or send message.

Primary problem of these approaches is the complexity, and the lack of assistance for forward engineering tools like code generation.

3. METHODOLOGY

The methodology consisted of two major parts. Firstly, the representation of design pattern catalog and secondly the automated code generation. Our goal was to generate an abstract code for design patterns. In order to achieve that, we need to represent design pattern catalog in a formal language that the machine can identify. This section describes our approach to solve these problems.

3.1 Meta-model of Design Pattern Catalog

Techniques based on meta-model consist in defining a set of meta-entities from which a design pattern description is obtained by composition of these entities using an instantiation link. This composition follows semantic rules fixed by the relations among meta-entities. A pattern meta-model does not capture a pattern in general but represents how it used in one or several specific cases. Pattern meta-model never produce models of patterns.
Meta-model (static structure of meta-entities) represented in Figure 1, mainly consists of `DesignPatternCatalog`, `Pattern`, `PEntity` and `PElement` classes (last 3 are abstract classes). Meta representations of design patterns are created by extending the `Pattern` abstract meta class. `Pattern` abstract meta class consists of many `PEntity` meta instances. `PEntity` meta abstract class is the super class for `PClass`, `PInterface` and `PAbstractClass` meta-entity representations of class, abstract class and interface. `PEntity` consists many `PElement` instances. `PElement` is the super class for `PAssociation`, `PMethodElement` and `PField`, meta-entity representations for an association, method and field. By extending `PMethodElement` abstract meta class, `PAbstractMethod` and `PMethod` meta-entities can be generated. Then by extending the `PMethod` meta-entity, `PDelegatingMethod` meta-entity (represents delegation) is generated.

3.2 Instantiation of Design Patterns Using Meta-model

Representation of a Design pattern using meta-entities consist two major steps as [2], i.e.

1. First specialize the meta-model to add all the structural and behavioral entities/elements needed constituents. If a new entity or element is needed for represent a design pattern, in other words, if the entities and elements of the meta-model are insufficient for a particular pattern, then by using generalization, we can add new entities or elements to the meta-model.

2. The second step involves instantiation of the design pattern built according to the generic meta-model semantics. The resulting model of the pattern is called an abstract model of the design pattern because it does not retain any information about the user’s application context. This abstract model corresponds to a reification of the design pattern and holds all the needed information related to the design pattern.
Meta-model diagram depicted in Figure 1 is enough to hold information of the abstract definition of Singleton, Abstract Factory, Composite, Proxy and Strategy patterns represented at [1]. In order to demonstrate the representation of patterns in meta-model approach, the Proxy Pattern is selected.

2.3 Proxy Design Pattern Representation Using Meta-model

Solution structure of the Proxy Pattern and its object structure at run time are depicted in Figures 2 and 3 respectively.
Figure 2: Structure of the Proxy design pattern.

Figure 3: Object structure at run time of the Proxy design pattern.

The following section illustrates the meta-model representation (meta-representation) of the proxy design pattern.

We can obtain a meta-instance (abstract) of proxy design pattern by instantiation the Proxy class and can retrieve an abstract instance of Proxy Pattern for the abstract structure of the proxy design pattern by calling the `getAbstractProxyPattern()` method. Following code segment shows how Proxy design pattern can be represented using meta-model.

```java
//Source file: C:/Proj-400/metamodel/patterns/Proxy.java
package patterns;

import patterns.metamodel.*;
import java.util.*;

public class Proxy extends Pattern {
    public Proxy() {
        super( "Proxy","Surrogate","Provide a surrogate or placeholder for another object to control access to it" );
    }
    public Proxy( String name,String otherNames,String intent ) {
        super( name,otherNames,intent );
    }

    /**
    Initialize Proxy pattern for abstract solution */
```
public static Pattern getAbstractProxyPattern()
{
    Pattern proxyPattern = new Proxy();
    // declare subject interface
    PInterface subject = new PInterface( "Subject","public" );
    // define request abstract method
    PMethodElement request = new PAbstractMethod( "Request","public",subject,null,new Vector() );
    // add request abstract method to the subject interface
    subject.addElement( request );
    // declare RealSubject class
    PClass realSubject = new PClass( "RealSubject","public" );
    realSubject.setImplementedInterface( subject );
    // declare Request method overrides the request method
    PElement realRequest = new PMethod( "request","public",realSubject,null,new Vector(),null,request );
    // add realRequest to the realSubject method
    realSubject.addElement( realRequest );
    // declare Proxy class
    PClass proxyClass = new PClass( "Proxy","public" );
    proxyClass.setImplementedInterface( subject );
    // declare request method overrides the request method of the subject
    Vector paraVec = new Vector();
    paraVec.add( new PParameter( "realSubject","RealSubject" ));
    PElement proxyRequest = new PDelegatingMethod( "request","public",proxyClass,null,paraVec,null,
      new PAssociation( "realSubject","public",proxyClass,realSubject ),realRequest,request );
    // add delegating method to the proxy class
    proxyClass.addElement( proxyRequest );
    // add all entities to the proxyPattern
    proxyPattern.addEntity( subject );
    proxyPattern.addEntity( realSubject );
    proxyPattern.addEntity( proxyClass );
    return proxyPattern;
}

2.4 Automated Code Generation

A design pattern only describes the solution to a particular design problem; it is not a code by itself. Our meta-model representation can be used to generate an abstract code for a pattern in object oriented programming languages such as Java and C++. This section describes how the proposed meta-model can be applied for automatic code generation for patterns.

Pattern abstract class consists of a PatternBuilder abstract class; separate concrete classes are created by extending PatternBuilder for different languages (JavaBuilder, CSharpBuilder, and CPPBuilder etc.), and are responsible for creating code for the pattern. Figure 4 shows the structure of the meta-model that can be used for code generation.
Builder classes build specified programming language code for the pattern for all entities and all elements of those entities. Resulting code includes implementation for all entities and elements declared in the pattern.

4. RESULTS

A prototype system was developed to test the applicability of the proposed framework for automatic code generation. Test results are presented here.

4.1 Generated Code for Proxy Design Pattern

```java
/** Java source code for interface Subject
 ******************************************/
public interface Subject {
    public void Request();
}

/** Java source code for class RealSubject
 ******************************************/
public class RealSubject implements Subject {
    public void request() {
        /* implementation notes */
    }
}

/** Java source code for class Proxy
 ******************************************/
public class Proxy implements Subject {
    public void request(RealSubject realSubject) {
        /* implementation notes */
        realSubject.request();
    }
}
```
5. CONCLUSION

The design pattern automation is a continuous process because a design pattern is actually a reasonably complex document describing a problem in a given context surrounded by various forces. Any solutions offered are qualified by various trade-offs of using certain designs and implementations over various contexts. Thus, a pattern is not just a solution structure consisting of a UML diagram, rather it is an essay for the designer to read, think and adapt. Therefore, complete design patterns automation is an impossible topic. However, to reduce the complexity, development time and implementation time, at least designers or programmers should be assisted by intelligent tools.

Abstract code doesn’t make any sense in real world development. Therefore, we need a tool that enables designers to do their application specific design on a pattern with maintaining pattern’s internal structure. By using a transformation technique, we can transform the abstract representation to application specific design with maintaining internal structure of the pattern. Designer should be able to add and remove functions, rename an existing function, add new entities to the pattern and able to customize it for application requirements. As an example in proxy design if designer adds a new public function for `RealProxy` class, then there should be a public function on `Proxy` to access that function. Tool should generate second function automatically and remove it if designer remove the first one, or rename it should the designer choose to rename the first one. That is, the tool is intelligent to maintain the consistency in the underlying design pattern instance. Research work on solving this particular issue is currently being carried out.

REFERENCES

5. Bernd-Uwe Pagel and Mario Winter. Towards Pattern-Based tools.
6. Andy Bulka, Design pattern automation.
7. James O Coplien, Coding Patterns.