Detecting Intent Aspects from Code to Apply Design Patterns in Refactoring: An Approach Towards a Refactoring Tool

D.Janaki Ram, J.Rajesh
{djram, rajesh }@cs.iitm.ernet.in
Distributed Object Systems Lab, Dept. of Computer Science & Engg.,
Indian Institute of Technology, Madras, India.

Keywords: *Design Pattern, Intent Aspect(IA), Predicate Templates, Refactoring, Rule Base, Facts Base*

1. EXTENDED ABSTRACT

1.1 Introduction

Software evolves as requirements change. Changes are inevitable, as users request new features, external interfaces evolve and designers better understand the key elements of the application. In order to adapt changes, the software should be designed and coded in such a way that it provides high degree of flexibility and reusability. There may exist software systems which exhibit very low quality in terms of reusablity, flexibility and extendability. This is where refactoring becomes significant. Refactoring is "the process of changing a software system in such a way that it does not alter the the external behavior of the code, yet improves it's internal structure and so quality"[1].

Refactoring has been gained much more attention in the object-oriented software development. Though there are several ways to refactor object-oriented software systems, refactoring using design patterns has been more of research interest. Design patterns are "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [2]. The Software systems developed using design patterns are more reusable and extendable. Hence, refactoring using design patterns leads to high quality software.

There are three distinct steps in refactoring process[3]:

1. detect when an application should be refactored
2. identify which refactoring should be applied
3. where to perform these refactorings

There are tools for automatically applying transformations[4],[5] (step3), once a design pattern is chosen. However, there are no tools to choose which design pattern to be applied and what are the program structures or elements to which the chosen design pattern is applied. In this paper, we address the refactoring issues such as, where design patterns can be applied in the code and which design patterns are to be chosen.

In our approach, we are primarily focusing on java code refactoring since java becomes one of the most popular object oriented language. We identify Intent Aspects(IA) by using TyRuBa(Type Rule Base)[6] rules or Prolog[7] rules and using java parser and facts generator from java source code. We define Intent Aspects as set of program structures or elements (e.g, class, method, etc.) which implies the applicability of a suitable design pattern.In this paper, we propose a technique to develop a tool, JIAD(Java based Intent Aspects Detector) that will automate the identification of IAs which helps to apply suitable design pattern while refactoring the java code. After that the transformation can be applied to the detected IAs to change them to chosen design pattern related code. This is an approach to automate the whole process (steps 1,2,3) of refactoring. This helps not only in rapid evolutionary software development, but also in developing high quality software systems.

1.2 Overview of the Approach

The Architectural Model of JIAD is shown in figure.1. In the figure, predicate templates are"definitions of the primitive rules up on which derived rules are built". Java Parser and Facts generator takes the java source code and the predicate templates as input and generates set of facts which are defined as per the predicate templates and stores them in facts base. The Java parser generator is similar to the the parser used in QJBrowser[8] with extended predicate templates. All other derived rules are formed by using the primitive rules. [9] provides mechanics section for manually refactoring the non-pattern based code. This is also considered while forming rules. For each design pattern, a set of rules have been formed by analyzing design pattern Intent and Applicability sections extensively, by analyzing the non-pattern based code and also by analyzing the metrics from [10]. The rules are stored in a data structure called design pattern rule base. These rules detect the IAs where the selected pattern can be applied. Each of these rules is matched against the facts from facts base by using TyRuBa or PROLOG. If the rule is satisfied, then the predicate variables are instantiated from the facts. The instantiated values (classes

SODA'03, January, 2004, Bangalore, India.

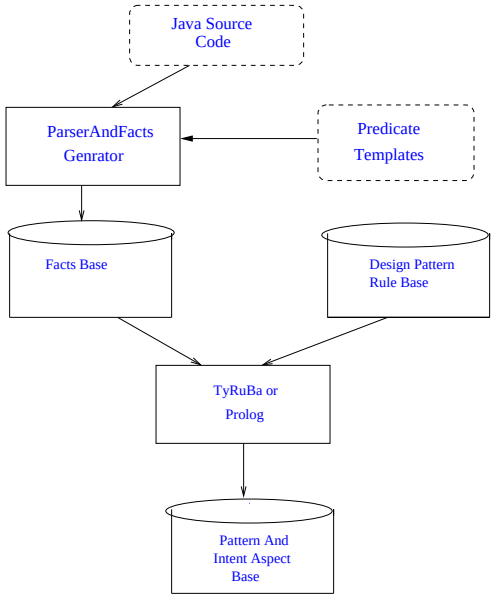

Figure 1: Proposed Architectural Model of JIAD

or methods or conditional statements, etc.,) of these predicate variables are stored in some temporary data structure. Finally, each design pattern transformation can be applied on the related set of IAs to refactor them.

1.3 Rules to Detect IAs

IAs for each pattern are identified by the following informal rules:

1. Abstract Factory:

   If a method has conditional bodies creating product objects through hard coded constructor statements which belong to different inheritance trees and depth of the inheritance trees is more then abstract factory pattern can be applied.

2. Builder:

   If a method is creating objects through hard coded constructor statements and these created objects together form a complex object by taking some other objects as parameters in their constructor statements or in their method calls then builder pattern can be applied.

3. Factory method:

   If a method doesn't satisfy the rules of abstract factory or builder and is creating an object by using hard coded constructor and if it also has some other statements or methods which belong to the same class then factory method pattern can be applied.

4. Prototype :

   If a method is creating objects through same hard coded constructor or the products created through constructors are in the same inheritance hierarchy then prototype pattern can be applied.

5. Singleton:

   If a class has all static members which are publicly accessible, or if it has single instance in the application then singleton pattern can be applied.

6. Decorator:

   If the number of methods that are overriding the methods and that are calling the overridden method of the super class is greater than the number of methods that are newly defined in the sub classes and that are being called from overridden method then decorator pattern can be applied.

7. Composite:

   If a class has 1:N recursive aggregation to itself and has an aggregation relationship to another leaf class then composite pattern can be applied.

8. State:

   Assume that, a method is simulating finite state machine(FSM) with netsted conditional block. If the nested conditional block satifies all the following conditions then state pattern can be applied.

   a) For all outer conditional(similar to input in FSM) expressions there are same inner conditional(similar to present state in FSM) expressions.

   b) All the inner conditional blocks should modify the inner conditional variable(state change in FSM).

9. Strategy:

   If a method doesn't satisfy the rule of state pattern and has more number of conditional blocks with more number of statements then strategy pattern can be applied.

10. Template Method:

    If a method is calling the method of the same class and the called method is not overridden in the derived classes then template method pattern can be applied.

Due to space limitation, the TyRuBa rules for Singleton and Template Methods are shown below:

```
Singleton(?C) :-
  ( class(?C),
  forall( context(?Member, ?C),
  modifier(?member, 'static'),
  modifier(?member, 'public'));
  instances(?C, ?Number),?Number==1.
  ) .

TemplateMethod(?C, ?Method, ?HMethod) :-
  ( HasMethod(?C,?Method),
  InvokingMethod(?Method, ?HMethod),
  HasMethod(?C, ?HMethod),
  not(IsOverridding(?HMethod)).
  ).
```

The descriptions for the above rules are given at the starting of this section.

1.4 Example

This section shows an example for detecting IAs to apply state pattern for the given code. The example code is shown below:

```
1. class FSM
2. {
3. void fsm()
4. {
5. switch(request)
6. { case "open": { switch(state)
7. { case "TCPE": { ... }
8. case "TCPL": { ... }
9. case "TCPC": { ... }
10. }
11. case "closed": { switch(state)
12. { case "TCPE": { ... }
13. case "TCPL": { ... }
14. case "TCPC": { ... }
15. }
16. case "Ackn": { switch(state)
17. { case "TCPE": { ... }
18. case "TCPL": { state=... }
19. case "TCPC": { ... }
20. }
21. }
22. }
23.
24.}
```

Fig2. An Example Code Fragment

As per the rule for state pattern mentioned above, the code simulates the FSM. The code fragment shown in Fig2 and predicate templates such as parentSwitch(StartLine, ParentID), caseConditionValue(SwitchID, VarName, VarValue, ParentCaseID), methodHas(MethodID, SwitchID, ClassID), etc, are given as input to ParserAndFacts generator. The ParserAndFacts generator parses the file and generates TyRuBa or Prolog facts as specified by the predicate templates. Some of the facts that the ParserAndFacts generator generates are shown below:

```
parentSwitch(6,5).
parentSwitch(16,5).
parentSwitch(11,5).
caseStmt(6,5,4).
caseStmt(11,5,4).
caseStmt(16,5,4).
caseConditionValue(6,'state','TCPE',6).
caseConditionValue(6,'state','TCPL',6).
caseConditionValue(6,'state','TCPC',6).
caseConditionValue(6,'state','TCPE',11).
caseConditionValue(6,'state','TCPE',11).
caseConditionValue(6,'state','TCPL',11).
caseConditionValue(6,'state','TCPC',16).
caseConditionValue(6,'state','TCPL',16).
caseConditionValue(6,'state','TCPC',16).
methodHas('fsm',5,'FSM').
```

Now, as mentioned in the rule of state pattern, for all outer case statements(lines 6,11,16) of the parent switch statement(5) of Fig2, TyRuBa or Prolog matcher verifies that all child case condition values("TCPE", "TCPL","TCPC") and variables(state) are same or not. If they are same which means rule is satisfied, then it returns the method name(fsm), class name(FSM) and part of the switch code fragment as IA. This part of the code should be transformed to pattern(state) oriented implementation.

1.5 Conclusions and Future Work

Currently, we have proposed rules for all creational patterns, few of the behavioural patterns and structural patterns. We need to implement these rules as per the proposed architecture. The proposed approach addresses the refactoring issues of where to apply and which refactoring should be applied by identifying the IAs and by choosing suitable design pattern. Finally, suitable transformations can be applied on the IAs according to the chosen design pattern. Thus, this enables rapid evolutionary software development and production of high quality software systems by making the code reusable and extendable.

2. REFERENCES

[1] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[2] E.Gamma, R.Helm, R.Johnson, and J. vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley, 1994.

[3] Tom Tourwe and Tom Mens. Identifying Refactoring Oppurtunities Using Logic Meta Programming. *7th European Conference on Software Maintenance and Reengineering*, page 91, March 2003.

[4] Don Batory and Lance Tokuda. Automated software evolution via design pattern transformations. Technical Report CS-TR-95-06, The University of Texas at Austin, Department of Computer Sciences, 1995.

[5] Mel O Cinneide. Automated Refactoring to Introduce Design Patterns . In *International Conference on Software Engineering*, pages 722–724. ACM Press,Limerick , June 2000.

[6] TyRuBa. http://tyruba.sourceforge.net.

[7] Wielemaker. Prolog Reference Manual. http://swi.psy.uva.nl/projects/SWI-Prolog.

[8] R.Rajagopalan and Volder. QJBrowser: A Query-Based Browser for Exploring Cross Cutting

Concerns in Code. *Technical Report, University of British Columbia*, 2002.

[9] Joshua Kerievsky. *Refactoring to Patterns*. Industrial Logic, Inc, 2002.

[10] Taichi Muraki and Motoshi saeki. Metrics for Applying GOF Design Patterns in Refactoring Process. *International WorkShop on Principles of Software Evolution(IWPSE)*, 2001.