# JIAD: A Tool to Infer Design Patterns in Refactoring

J Rajesh
Distributed and Object Systems Lab
Dept. of Computer Science & Engg.
Indian Institute of Technology Madras
Chennai, India
rajesh@cs.iitm.ernet.in

D Janakiram
Distributed and Object Systems Lab
Dept. of Computer Science & Engg.
Indian Institute of Technology Madras
Chennai, India
djram@cs.iitm.ernet.in

## ABSTRACT

*Refactoring in object-orientation has gained increased attention due to its ability to improve design quality. Refactoring using design patterns (DPs) leads to production of high quality software systems. Although numerous tools related to refactoring exist, only a few of them apply design patterns in refactoring. Even these do not clearly specify where refactoring can be applied and when to apply appropriate design patterns.*

*In this paper, we propose a tool, JIAD (Java based Intent Aspects Detector) which addresses the refactoring issues such as, the scope for applying DPs in the code and the appropriate selection of DPs. The tool automates the identification of Intent-Aspects (IAs) which helps in applying suitable design patterns while refactoring the Java code. By automating the process of identifying IAs, the whole process of refactoring using DPs can be automated which enables rapid development of software systems. Also, the tool minimizes the number of possible errors while inferring the suitable DPs to apply refactoring. Our approach primarily focuses on Java code refactoring using declarative programming and AspectJ compiler. Finally, the tool is validated using two applications namely, JLex and Java2Prolog written in Java.*

## Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, Enhancement—*restructuring, reverse engineering, and reengineering*; D.1.2 [**Programming Techniques**]: Automatic Programming—*program verification, program transformation*; D.1.6 [**Programming Techniques**]: Logic Programming; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*object-oriented programming*

## General Terms

Design, Languages, Reliability, Verification


*PPDP'04,* August 24–26, 2004, Verona, Italy.


**Keywords:** *Design Pattern (DP), Declarative Programming, Object-Oriented Systems, Intent-Aspects (IAs), Prolog, Predicate-Templates, Refactoring, Rule-Base, Facts-Base*

## 1. INTRODUCTION

Software is not stable. Often it undergoes constant changes. Changes are inevitable, as users request new features, external interfaces evolve and designers understand the key elements of the application better. In order to adapt to changes, the software should be designed and coded in such a way that it provides high degree of malleability (high degree of flexibility, reusability and extendability). There may exist software systems which exhibit very low quality in terms of reusability, 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 external behavior of the code, yet improves it's internal structure and hence quality"[17].

Refactoring has gained more attention in object-oriented software development. Though there are several ways to refactor object-oriented software systems, refactoring using design patterns[1] has been more of research and practical interest. Design patterns are "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [8]. 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[21]:

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

There are tools[24, 10, 25] that help in refactoring the code without introducing design patterns. There are also few tools for automatically applying transformations[5, 18], once a design pattern is chosen manually. However, to the best of our knowledge 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 to be applied.

In this paper, we propose a technique to identify Intent-Aspects (IAs) from Java code. IAs are "a set of program elements or structures (e.g, classes, methods, conditional blocks, etc), which implies the applicability of a suitable

[1]In the remainder of this paper, design patterns and patterns are used interchangeably.

design pattern through the interactions of the program elements"[4]. The IAs are identified with the help of declarative programming rules. A collection of facts about Java code is generated with the help of Java code parser and facts generator. Each pattern has a set of rules and these rules are verified against the collection of facts. If any one of the rules is satisfied, then it extracts the IAs from the query variables which are instantiated from the collection of facts by the query engine. The transformation can be applied on the detected IAs to modify them by introducing the chosen design pattern. This approach to automate the whole process (steps 1,2,3) of refactoring has been proposed in the paper. This helps not only in rapid evolutionary software development, but also in developing high quality software systems.

The rest of the paper is organized as follows. Section 2 describes the motivation for the tool, JIAD; Section 3 describes the architectural overview of the proposed tool, JIAD; Section 4 introduces the declarative meta programming and predicate-templates which play a key role in detecting IAs; Section 5 describes the informal description of the rules for each design pattern which detect IAs and the approach of detecting the IAs with the help of an example; Section 6 describes how the tool is implemented; Section 7 gives a case study for proof of concept; Section 8 discusses prior work related to refactoring and tool support for refactoring; Finally in section 9, we present some conclusions and future work.

## 2. MOTIVATION

Design patterns are not widely used by software designers, due to lack of well-defined, systematic approaches for using them in the design process. However, software systems developed using design patterns yield higher quality (like extendibility, reusability, flexibility, etc.) than those developed without using them. Also, these software systems (with design patterns) are easily adaptable to changes. In order to transform existing low quality software systems into high quality ones, it is advisable to have design patterns as refactoring targets since they are quality designs.

Tool support for refactoring helps to speed up software development with minimum cost. The existing refactoring tools need to be provided with the information about the part of the code which needs to be refactored and the design patterns that should be applied. This requires manual observation of the code. Once the programmer manually analyzes the code extensively, he chooses that part of the code which needs to be refactored and a suitable design pattern. This is very tedious, error prone and incurs effort and cost. Automatic support to infer which part of the code needs to be refactored and which design pattern is suitable to apply on the inferred code, minimizes errors and cost.

## 3. DESCRIPTION OF JIAD ARCHITECTURE MODEL

The architectural model of JIAD is shown in Figure 1. In the figure, predicate-templates are *definitions of the primitive clauses about an object-oriented language upon which rules are built.* Java Parser and Facts Generator takes the Java source code and the predicate-templates as input and generates a set of facts which are defined as per the predicate-templates and stores them in facts-base. The ParserAndFacts generator is similar to the parser used in QJBrowser [19] with extended predicate-templates. All other derived rules are formed by using the clauses and primitive rules. [14] provides manual refactoring methods for 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[8] extensively, by analyzing the non-pattern based code and also by analyzing the metrics from [20]. These set of rules for all design patterns is called rule-base. These rules detect the IAs where the selected pattern can be applied. Each of these rules from the rule-base is matched against the facts from facts-base by using either Prolog[23] or TyRuBa[2], declarative languages. Facts-base is defined as *collection of inter-related facts corresponding to a project or a program.* If the rule is satisfied, then the predicate variables are instantiated from the facts. The instantiated values (classes or methods or conditional statements, etc.,) of these predicate variables are stored in some temporary data structure. The tool displays part of the code and the design pattern that can be applied to it. Finally, each design pattern transformation can be applied on the related set of IAs to refactor them.

## 4. DECLARATIVE META PROGRAMMING AND PREDICATE-TEMPLATES

### 4.1 Declarative Meta Programming

Declarative Meta Programming (DMP) is defined as "the use of a declarative programming language at a meta level to reason about and manipulate programs built in some underlying base language"[13]. Declarative meta programming[15] is currently being investigated as a technique to support state-of-the-art software development. It is based on a tight dependence between object-oriented language and declarative language. This makes it possible to reason about object-oriented programs in an intuitive way. In this paper, DMP is used to easily extract the information from source code and verify it at a meta level. In this paper, the functionality of declarative meta programming is achieved by using the features of AspectJ compiler[3] and Prolog or TyRuBa. Prolog and TyRuBa are two such examples for declarative languages. The inference rules used in this paper are based on Prolog. The same inference rules can also be used with TyRuBa with small variations in syntax.

### 4.2 Predicate-Templates

A Prolog program consists of clauses. Each clause is either a fact or a rule. A predicate is a collection of clauses with heads that match each other. To generate the basic facts from the program, predicate-templates are defined. Predicate-templates are *definitions of the primitive clauses about an object-oriented language upon which rules are built.* The predicate-templates that follow in this section are related to Java, since we are focusing on code refactoring in Java. Table 1 gives brief description about predicate-templates which are necessary to illustrate the example in section 5.2. There are several such predicate-templates, which are explained in detail in Appendix B.

The predicate-templates are used by AspectJ compiler and facts generator. The predicate-templates shown in Ta-

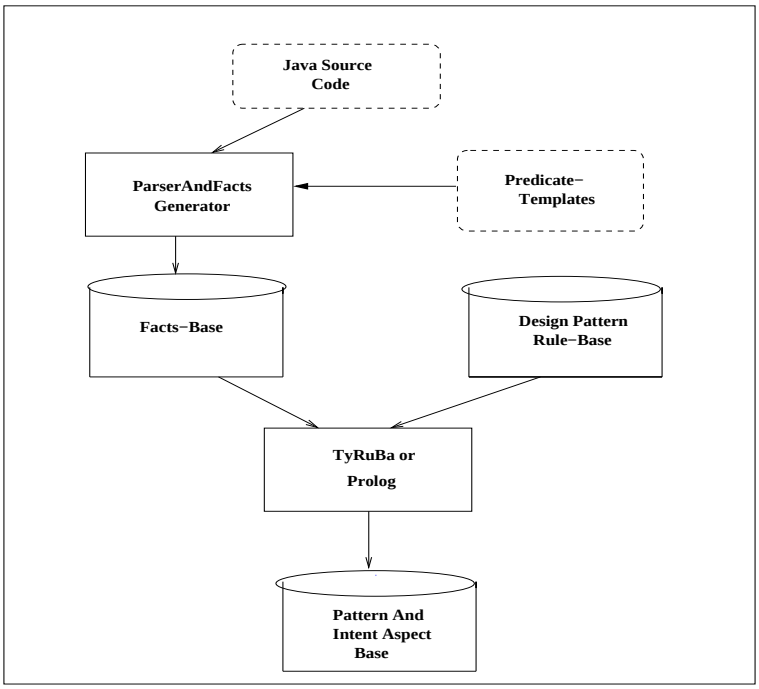


**Figure 1: Architectural Model of JIAD**

**Table 1: Predicate-Templates to Generate Prolog/TyRuBa Facts**

| Clause | Description |
|---|---|
| class(*Class*[2]) | *Class* is a class. This is useful to reason about class and its relations to other classes and interfaces, etc. |
| interface(*Interface*) | *Interface* is an interface. This is useful to reason about inheritance hierarchy using interfaces. |
| method(*Method*) | *Method* is a method of some class. The *Method* contains both method name and class or interface in which it is a member. This is useful to reason about inter and intra class interactions. |
| shortname(*Member*, *ShortMemberName*) | class or interface member *Member* has short name *ShortMemberName*. This is useful to reason about methods and fields of a class or interface. |
| arg(*Method*, *Argument1*) | method *Method* has argument *Argument1*. The same method can have number of arguments in which case it has number of arg clauses each with different argument. This Clause is useful to reason about overridden methods, method signature, object coupling through method calls. |
| context(*Member*, *X*) | *Member* is a member of class or interface *X*. This is useful to reason about members of a class. |
| extends(*Derived*, *Base*) | class *Derived* is derived from class *Base* or interface *Derived* is derived from interface *Base*. This is useful to reason about inheritance hierarchy, depth of the inheritance tree. |
| implements(*Derived*,*Base*) | class *Derived* is derived from interface *Base*. This is similar to extends Clause as mentioned above except that it reasons about multiple inheritance. |
| callinfo(*Caller*, *Callee*, *CodeInfo*) | method *Caller* calls the method *Callee* in the code at *CodeInfo*. This is useful to reason about the call sequence across classes. |

[2]In this table and in Appendix B, Prolog variables are denoted with italic font style. All Prolog variables start with capital letters.

ble 1 are used to generate facts from the Java code. Knowledge about Java source code is captured in the form of facts. These facts are used to reason about design pattern IAs.

# 5. DETECTING INTENT-ASPECTS

This section gives an informal description of rules to detect IAs for each design pattern and how the IAs can be detected using Prolog. Since the rules for identifying IAs are based on the intent and applicability sections of the design patterns, they are summarized in Appendix A. The formal rule (Prolog rule) for Decorator pattern with an example is explained in subsection 5.2. Due to space limitation formal rules for other patterns are not shown.

## 5.1 Description of Rules to Detect Intent-Aspects for Design Patterns

Intent-Aspects (IAs) for each pattern are identified by the following informal rules:

1. **Abstract Factory:**

   If all the following rules are satisfied then Abstract Factory pattern can be applied:

   - Product objects are created in conditional blocks of a method through hard-coded constructor statements.
   - Product objects belong to different inheritance trees.
   - Number of derived classes in the inheritance trees are equal at the same level.
   - Depth[3]of the inheritance trees is greater than or equal to 1.

2. **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.

3. **Composite:**

   If all the following rules are satisfied then Composite pattern can be applied.

   - A class has an 1:N (one to many) recursive aggregation relationship to itself. Call it as composite class.
   - The composite class has aggregation relationship to at least one class. Call these classes as leaves.
   - All the classes, composite and leaves, have at least one method in common.

4. **Decorator:**

   If all the following rules are satisfied then Decorator pattern can be applied.

   - There exists an inheritance tree.
   - There exist overridden methods which call at least one newly defined method and also call the same overridden method of the super class.
   - For every newly defined method, it is called by the every overridden method.
   - The list of classes from which newly defined methods are called through overridden methods is equal to the derived class list.

5. **Template Method:**

   If all the following rules are satisfied then the Template Method pattern can be applied.

   - There exist an inheritance tree.
   - There is at least one method in all the derived classes of this inheritance tree in such a way that there is a common part across all the methods.

The IAs for each pattern is generated if the corresponding rule is satisfied. Description of rules for remaining patterns are not provided here due to space limitations.

## 5.2 Detecting Intent-Aspects Using Prolog

In this section, we explain how low quality design with inheritance hierarchy depicted in Figure 2 is translated into high quality design by the application of Decorator design pattern with the help of Prolog rules.

Figure 2 illustrates an inheritance tree that contains many overriding methods and newly defined methods. For example, the method M1 in the class Derived1 overrides the M1 that is defined in the super class Base and it sends a method invocation to M1 of Base. The characteristics of the tree are as follows:

1. Many occurrences of overriding methods (4 occurrences) strongly connect the sub classes to each other in the tree. In particular, since many of them invoke the same methods defined in their super classes, the connectivity and dependency among the classes are much stronger. This characteristic makes the modification of inheritance structure more difficult.

2. Since the tree includes many newly defined methods (6 methods) that are not overridden by any method and many of them are scattered across the classes (6 methods are scattered across 5 classes), it would be difficult for us to add new responsibilities to the tree. To change the inheritance structure, we need to explore all of the sub classes and check newly introduced methods.

To overcome these problems, the design needs to be refactored. By using the Decorator design pattern the same functionality can be achieved using the design shown in Figure 3. Given a Java code which has the structure shown in Figure 2 as part of the code, the tool identifies the IAs such as classes and methods which needs to be refactored and the design pattern (Decorator)that can be applied to improve the quality while preserving the functionality. This process is illustrated in the following lines with the help of Prolog rules shown in Table 2. These rules help in identifying IAs of Decorator design pattern.

The tool takes the Java code which has the structure shown in Figure 2 as part of the code. By using the predicate-templates explained in sub-section 4.2, the tool generates the facts-base which is used to reason about the Java code. A few facts that belong to the facts-base are shown in Table

[3]Depth is defined as maximum number of edges from the base class to the leaf of the inheritance tree.

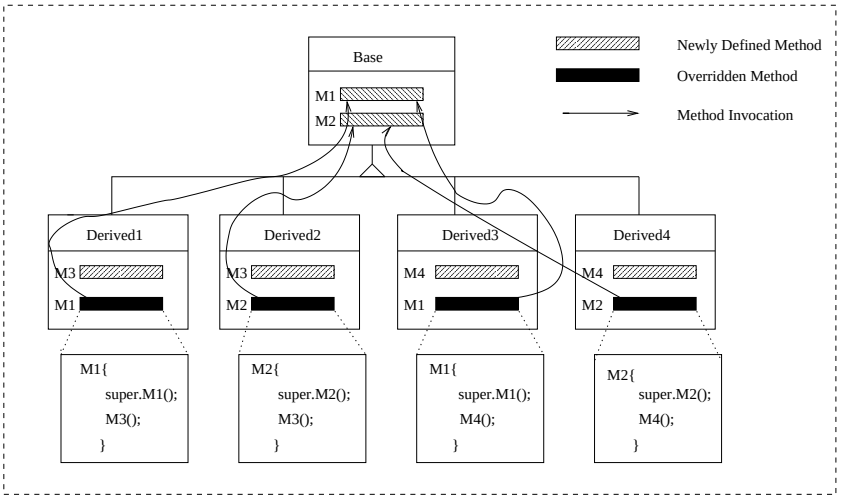


**Figure 2: Example Design of Low Quality: Inheritance Tree**

**Table 2: Prolog Rules to Detect IAs for Decorator Pattern**

**decorator(*BaseClass, OverriddenMethodList, NewlyDefinedMethodList*)** :-
derived_class_list(*DerivedClassList, BaseClass*),
list_of_new_and_over_methods(*BaseClass, OverriddenMethodList, NewlyDefinedMethodList*),
setof(*ShortMethodNames*,
(member(*Method, OverriddenMethodList*), shortname(*Method, ShortMethodName*)),
*ShortNameList*),
forall(member(*NewMethod, NewlyDefinedMethodList*),
( forall(member(*OMethod, ShortnameList*),
(callinfo(*XM, NewMethod,_*), shortname(*XM, OMethod*),member(*XM, OverriddenMethodList*))) ) ),
setof(*Class*, (member(*OverMethod, OverriddenMethodList*), context(*OverMethod, Class*)),
*DClassList*),
*DerivedClassList*==*DClassList*.

**list_of_new_and_over_methods(*Base, OverriddenMethodList,NewlyDefinedMethodList*)**:-
(class(*Base*), extends(*Base*,'java.lang.Object'); interface(*Base*),not(extends(*Base,X*)) ),
setof(*NewMethod1*,
*DerivedOverMethod1*^*DerivedClass1*
^new_methods_called_from_overMethod(*Base, DerivedOverMethod1,NewMethod1,DerivedClass1*),
*NewlyDefinedMethodList*),
not(*NewlyDefinedMethodList*==[]),
setof(*DerivedOverMethod*,
*NewMethod*^*DerivedClass*
^new_methods_called_from_overMethod(*Base, DerivedOverMethod,NewMethod,DerivedClass*),
*OverriddenMethodList*),
not(*OverriddenMethodList*==[]).

**new_methods_called_from_overMethod( *BaseClass, DerivedOverMethod, NewMethod, DerivedClass*)** :-
(extends(*DerivedClass,BaseClass*); implements(*DerivedClass,BaseClass*)),
class(*DerivedClass*), not(*DerivedClass*==*BaseClass*),
method(*BaseOverMethod*), context(*BaseOverMethod, BaseClass*),
shortname(*BaseOverMethod, Method*), not( *Method*=='new'),
method(*DerivedOverMethod*), context(*DerivedOverMethod, DerivedClass*),
shortname(*DerivedOverMethod, OverMethod*),
*Method*==*OverMethod*, matchArgList(*BaseOverMethod, DerivedOverMethod*),
callinfo(*DerivedOverMethod, BaseOverMethod,_*),
callinfo(*DerivedOverMethod, NewMethod, _*),
shortname(*NewMethod, NM*), not(*Method*==*NM*), not(isOverridden(*NM*)).

3. The facts-base is used by the rules[4] shown in Table 2. Because of space limitation, only minimal set of facts are shown, which are enough to illustrate the above example.

In Table 2, the base rule **decorator** uses the **list_of_new_and_over_methods** and **new_methods_called_from_overMethod** rules. The predicate variable BaseClass holds the base class of the inheritance hierarchy, the OverriddeMethodList variable holds the list of overridden methods which call the newly defined method and also call the same overridden method of the base class and the NewlyDefinedMethodList variable holds the list of newly defined methods that are called by the methods in the OverriddenMethodList. The methods in the OverriddenMethodList have both method name and its context (derived class name). Instances of these three predicate variables form the IAs for the Decorator pattern.

The **new_methods_called_from_overMethod** rule is satisfied if there are overridden methods which call newly defined methods and also call the same overridden method of the base class and the newly defined method should not belong to the base class. This rule instantiates the NewMethod variable for every newly defined method. For the above example, instantiated values of the NewMethod variable are shown below:
*NewMethod*= **M3(), M4()**.

The **list_of_new_and_over_methods** rule is satisfied if it satisfies the new_methods_called_from_overMethod rule. This rule instantiates the *NewlyDefinedMethodList* variable and *OverriddenMethodList* variable for every inheritance tree. Predicate variables *NewlyDefinedMethodList* and *OverriddenMethodList* are instantiated with the following values:

*NewlyDefinedMethodList*=[**M3(), M4()**].

*OverriddenMethodList*=[**M1(), M2()**].

In the above two lists, only method names are shown. The actual list contains methods along with their class names in which these methods are defined.

The **decorator** rule checks for set of derived classes of a class(base class). If there are derived classes then the derived_class_list predicate is satisfied. Now, the variable *DerivedClassList* holds the following list:
*DerivedClassList*=[**Derived1, Derived2, Derived3, Derived4**]. The **derived_class_list** rule finds the list of derived classes in an inheritance tree. Its Prolog definition is shown in Appendix C.

Next, it checks for the **list_of_new_and_over_methods** predicate. If the predicate is satisfied, it instantiates the predicate variables. Next, it verifies that for every newly defined method in *NewlyDefinedMethodList*, it is called by every method in *OverriddenMethodList*. This is true for the above example.

Next, it finds the list of classes from which the newly defined methods are called and call this list as DClassList. So, the DClassList for the above example is,
*DClassList*=[**Derived1, Derived2, Derived3, Derived4**].

Finally, it checks whether the *DClassList* is same as the *DerivedClassList*. For the above example, the *DClassList* and *DerivedClassList* are same.

Thus, the Decorator pattern rule (**decorator**) is satisfied and IAs for the above example are shown below:
*BaseClass*=[**Base**], *DerivedClassList*, *NewlyDefinedMethodList* with methods and their classes and *OverriddenMethodList* with methods and their classes.

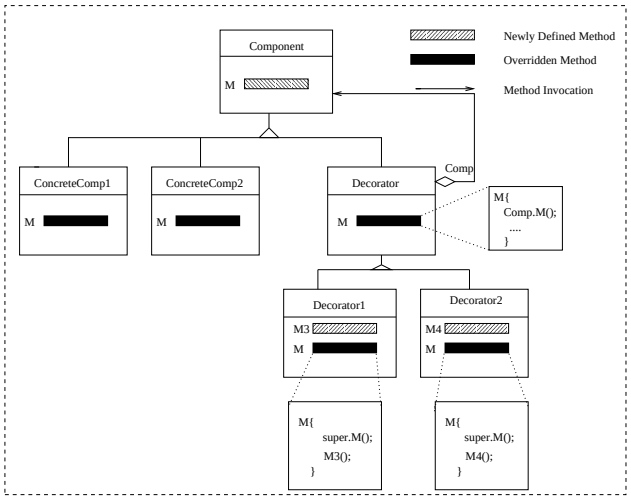


**Figure 3: Design with Decorator Pattern**

These IAs and the pattern name are used by the automatic transformation tools to transform the design shown in Figure 2 to high quality (more extendibility, flexibility) design shown in Figure 3 by introducing the Decorator pattern.The design shown in Figure 3 overcomes the problems mentioned in this section by bringing the newly defined methods together into a class and weaken the connectivity among the classes so that the design can be easily extendable while adding new responsibilities.

Rules for all the design patterns are stored in the rule-base and all of them are applied on the facts-base simultaneously. In the above example, only Decorator pattern rule is satisfied. Due to space limitation, rules for other patterns are not explained here. Currently, we have proposed rules for all creational patterns, few of the behavioral patterns and structural patterns.

# 6. IMPLEMENTATION

To refactor the Java programs, it is necessary to analyze its source code. The tool uses AspectJ compiler to parse the source code which constructs its parse tree or abstract syntax tree. The facts-base generator is used to traverse the parse tree thereby generating Prolog or TyRuBa facts according to predicate-templates explained in sub-section 4.2. The facts-base generator is written in Java. The Rule-base has a set of rules for each design pattern. These rules are written in Prolog. The same rules can be used for TyRuBa with minor changes. These rules detect IAs which need to be refactored with the corresponding design pattern. Each pattern has a set of rules which detect possible IAs. These rules can be extended as and when more possibilities of implementing the same functionality are identified. The rules are matched against the facts-base. Finally, the tool displays part of the code corresponding to IAs and its pattern name. The part of the code that corresponds to the intent-aspect and its design pattern name are given as input to the automatic transformation tool[5].

[4]Prolog rules are treated as formal rules.

**Table 3: Facts in the Facts-base of the Java Code which has the structure shown in Figure 2**

```
class('Base').
extends('Base','java.lang.Object').
method('Base.M1()').
shortname('Base.M1()','M1').
context('Base.M1()','Base').
method('Base.M2()').
return('Base.M2()','void').
shortname('Base.M2()','M2').
context('Base.M2()','Base').
class('Derived1').
extends('Derived1','Base').
method('Derived1.M1()').
return('Derived1.M1()','void').
shortname('Derived1.M1()','M1').
context('Derived1.M1()','Derived1').
callinfo('Derived1.M1()','Base.M1()',
   ['/.automount/parijatham/export/rajesh/QJBROWSER/EXAMPLE/paper_decorator.java',9,9]).
callinfo('Derived1.M1()','Derived1.M3()',
   ['/.automount/parijatham/export/rajesh/QJBROWSER/EXAMPLE/paper_decorator.java',10,10]).
method('Derived1.M3()').
shortname('Derived1.M3()','M3').
context('Derived1.M3()','Derived1').
class('Derived2').
extends('Derived2','Base').
method('Derived2.M2()').
shortname('Derived2.M2()','M2').
context('Derived2.M2()','Derived2').
callinfo('Derived2.M2()','Base.M2()',
   ['/.automount/parijatham/export/rajesh/QJBROWSER/EXAMPLE/paper_decorator.java',24,24]).
callinfo('Derived2.M2()','Derived1.M3()',
   ['/.automount/parijatham/export/rajesh/QJBROWSER/EXAMPLE/paper_decorator.java',25,25]).
method('Derived2.M4()').
shortname('Derived2.M4()','M4').
context('Derived2.M4()','Derived2').
class('Derived3').
extends('Derived3','Base').
method('Derived3.M1()').
shortname('Derived3.M1()','M1').
context('Derived3.M1()','Derived3').
callinfo('Derived3.M1()','Base.M1()',
   ['/.automount/parijatham/export/rajesh/QJBROWSER/EXAMPLE/paper_decorator.java',39,39]).
callinfo('Derived3.M1()','Derived2.M4()',
   ['/.automount/parijatham/export/rajesh/QJBROWSER/EXAMPLE/paper_decorator.java',40,40]).
class('Derived4').
extends('Derived4','Base').
method('Derived4.M2()').
shortname('Derived4.M2()','M2').
context('Derived4.M2()','Derived4').
callinfo('Derived4.M2()','Base.M2()',
   ['/.automount/parijatham/export/rajesh/QJBROWSER/EXAMPLE/paper_decorator.java',53,53]).
callinfo('Derived4.M2()','Derived2.M4()',
   ['/.automount/parijatham/export/rajesh/QJBROWSER/EXAMPLE/paper_decorator.java',54,54]).
class('decorat').
extends('decorat','java.lang.Object').
shortname('decorat','decorat').
method('decorat.main(String[])').
arg('decorat.main(String[])','java.lang.String[]').
shortname('decorat.main(String[])','main').
context('decorat.main(String[])','decorat').
code('decorat.main(String[])',
   ['/.automount/parijatham/export/rajesh/QJBROWSER/EXAMPLE/paper_decorator.java',61,66]).
creates('decorat.main(String[])','Derived1').
```

## 7. CASE STUDY

As a proof of concept, the tool has been tested on JLex[9]. The JLex utility is based upon the Lex lexical analyzer generator model. Lex is a lexical analyzer generator for the UNIX operating system, targeted to the C programming language. JLex takes a specification file similar to that accepted by Lex, then creates a Java source file for the corresponding lexical analyzer. JLex has total of 21 classes and 1 interface. It does not contain any inheritance trees. JIAD identifies IAs for 6 instances of Factory Method, 4 instances of Singleton and 2 instances of Strategy pattern. The IAs for 6 instances of Factory Method patterns are shown in Table 4. Each row in Table 4 is an IA corresponding to an instance of Factory Method pattern.

**Table 4: IAs for Factory Method Design Pattern**

| Class | Method |
|---|---|
| JLex.SparseBitSet | JLex.CMakeNfa.machine, JLex.CMakeNfa.term |
| JLex.CSet | JLex.CMakeNfa.machine |
| JLex.CAccept | JLex.CMakeNfa.machine |
| JLex.CBunch | JLex.CNfa2Dfa.make_dtrans |
| JLex.CDTrans | JLex.CNfa2Dfa.make_dtrans |
| JLex.CInput | JLex.CLexGengetState |

In Table 4, first row indicates that Factory Method pattern can be applied in the creation of product class **JLex.SparseBitSet** which is created in the methods **JLex.CMakeNfa.machine**, **JLex.CMakeNfa.term** through hard-coded constructor statement. This IA and Factory Method pattern name are given as inputs to the automatic transformer[5] to transform the code according to the Factory Method pattern related code.

IAs for 4 instances of Singleton pattern are shown in Table 5. Each row in Table 5 is an IA corresponding to an instance of Singleton pattern.

**Table 5: IAs for Singleton Design Pattern**

| Class |
|---|
| JLex.CNfa2Dfa |
| JLex.CMakeNfa |
| JLex.CSimplifyNfa |
| JLex.CEmit |

In Table 5, first row indicates that Singleton pattern can be applied to the class **CNfa2Dfa** since there exist single instance of the class in the application.

IAs for 2 instances of Strategy pattern are shown in Table 6. Each row in Table 6 is an IA corresponding to an instance of Strategy pattern.

**Table 6: IAs for Strategy Design Pattern**

| Class | Method | Case Block Line |
|---|---|---|
| JLex.CLexGen | userDeclare | 5313, 5379, 5425 |
| JLex.CLexGen | expandEscape | 6538, 6558, 6559 |

In Table 6, the first row indicates that Strategy pattern can be applied at the case blocks, starting at lines **5313, 5379, 5425** in the method **userDeclare** of class **JLex.CLexGen**. All the case blocks belong to the same switch statment.

Another case study has been performed on Java2Prolog[16] tool. Java2Prolog takes Java source code and generates Prolog facts related to class names, method names, inheritance hierarchies etc. It has a total of 100 classes and 3 interfaces with 4 inheritance hierarchies (trees). The characteristics: depth, width, of the inheritance trees are shown in Table 7. In Table 7, first row indicates that the depth and width of inheritance tree 1 (Tree ID) is 3 and 65 respectively. JIAD identifies IAs for 1 instance of Abstract Factory, 2 instances of Factory Method, 9 instances of Singleton, 1 instance of Strategy and 1 instance of Template Method pattern.

**Table 7: Characteristics of Inheritance Trees in Java2Prolog**

| Tree ID | Depth | Width |
|---|---|---|
| 1 | 3 | 65 |
| 2 | 1 | 3 |
| 3 | 1 | 2 |
| 4 | 1 | 2 |

We have manually inspected the source code and inferred some patterns. The patterns that are inferred manually coincide with the patterns inferred using the tool.

## 8. RELATED WORK

Refactoring[22][17] has been investigated by many researchers as a promising technique to improve the quality of the programs through behavior preserving program transformation. But, refactoring has gained more attention in object-oriented programming because it is well suited for evolutionary software development. Griswold[24] proposed a tool to apply refactorings and ensure that the meaning of the code was left unchanged by the refactoring.

A number of more recent tools also support refactoring: the Smalltalk Refactoring Browser[6], which automatically performs a set of refactorings on Smalltalk code. The IntelliJ Renamer tool[1], which supports renaming of packages, variables, etc. and moving of packages and classes for Java. Tokuda[5] shows that a typical system can evolve significantly faster and cheaper via refactoring and an automated introduction of the design pattern via the manual scripts. Roberts[7] discusses analysis to support refactoring, especially those defined by Opdyke[22]. Roberts defines postconditions that hold after a refactoring is applied. Cinneide[18] proposed a set of mini patterns and corresponding mini transformations that can deal with various refactorings. Emden and Moonen[10] developed a prototype tool that detect code smells in Java programs. Kataoka et al.'s[25] developed an invariant pattern matcher for several common refactorings and applied it to an existing Java code base. Tourwe and Mens[21] proposed an approach to show how automated support can be provided for identifying refactoring opportunities in Smalltalk.

None of these tools address the problem of automatically inferring the code that needs to be refactored and the appropriate design patterns that can be applied.

## 9. CONCLUSIONS AND FUTURE WORK

We have developed a prototype tool to detect IAs from Java code to apply design patterns in refactoring. The

predicate-templates which play a key role in generating facts-base are introduced. Prolog/TyRuBa rules to detect IAs of each design pattern are defined. The proposed tool addresses the refactoring issues of where to apply and which refactoring (design pattern) to be applied. It is done by detecting the IAs and by choosing a suitable design pattern. Automatic transformation can easily be performed on the IAs according to the chosen design pattern. The tool also identifies the interaction across IAs so that the same class or interface may be part of more than one pattern. This helps in capturing pattern interactions. By introducing the design patterns in the refactored code, designs can easily be modified according to the chosen design patterns. This helps in accommodating changes at the design level. The tool also minimizes manual errors. Thus, this enables rapid evolutionary software development by making the code reusable and extendable.

There may be number of ways of implementing same functionality. The rules to detect IAs are developed after examining the implementations done by programmers who do not have practice of using design patterns. These rules can be made much more accurate by analyzing different implementations for the same functionality. Sometimes design patterns introduce complexity. Therefore, before applying the transformation process, the impact of pattern oriented refactoring on quality attributes[11] needs to be estimated. Currently, we are proposing metrics to measure the quality attributes in terms of change. The code that is generated using automatic transformations[5][18] does not preserve pattern structure. Due to this, maintenance of software systems becomes very difficult as the software evolves. Glue patterns proposed in [12] implement design patterns while preserving the pattern structure. The automatic transformations using glue patterns need to be developed and it needs to be integrated with JIAD. This will make the entire refactoring process automatic, at the same time making maintenance simpler by preserving the pattern structure in the code.

## 10. REFERENCES


[1] IntelliJ . http://www.intellij.com.

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

[3] RajiNarayan. AspectJ Compiler. http://cvs.sourceforge.net/ viewcvs.py/ tyruba/ frontend/ org/aspectj/.

[4] D. Janaki Ram and J. Rajesh. Detecting Intent Aspects from Code to Apply Design Patterns in Refactoring: An Approach Towards a Refactoring Tool. In *Proceedings of 2nd Workshop on Software Design and Architecture(SODA)04*, Jan. 2004.

[5] 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.

[6] Don Roberts and Jhon Brant and Ralph Johnson . A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, pages 253–263, 1997.

[7] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, Univerisity of Illinois at Urbana-Champaign, June 1999.

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

[9] Elliot Berki. JLex: A lexical analyzer generator for Java. http://www.cs.princeton.edu/ãppel/modern/ java/JLex/.

[10] Eva van Emden and Leon Moonen. Java Quality Assurance by Detecting Code Smells. In *Proceedings of the 9th Working Conference on Reverse Engineering*. IEEE Computer Society Press, Nov. 2002.

[11] Jagdish Bansiya, Carl G. Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[12] Janaki Ram D, Dwivedi R.A., and Ongole Ramakrishna. An Implementation Mechanism for Design Patterns. *ACM Software Engineering Notes.*, 23(7):52–56, 1998.

[13] Johan Brichau and Kim Mens. Declarative Meta Programming. http://prog.vub.ac.be/research/DMP/.

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

[15] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.

[16] Marius Scurtescu. Java Program Representation and Manipulation in Prolog. Master's thesis, SFU_CS_School, 2000.

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

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

[19] R.Rajagopalan and Kris De Volder. QJBrowser: A Query-Based Browser for Exploring Cross Cutting Concerns in Code. *Technical Report, University of British Columbia*, 2002.

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

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

[22] W.F. Opdyke. *Refactoring Object-Oriented Frame-Works*. PhD thesis, Univerisity of Illinois at Urbana-Champaign, Computer Sciences Department, 1992.

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

[24] William G. Griswold and David Notkin. Automated Assistance for Program Restructuring. *ACM Trans. Softw. Eng. Methodol.*, 2(3):228–269, 1993.

[25] Yoshio Kataoka and Michael D. Ernst and William G. Griswold and David Notkin. Automated support for program refactoring using invariants. In *ICSM*, pages 736–743. IEEE Computer Society Press, 2001.

# APPENDIX

## A. DESCRIPTION OF GOF DESIGN PATTERNS

This appendix describes intent and applicability of GOF design patterns that are explained in sub-section5.1. Since the rules for identifying IAs are based on the intent and applicability sections of design patterns, only these two sections for each of them are explained below:

1. **Abstract Factory:**
   - **Intent:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
   - **Applicability:** It is used when a family of related product objects is designed to be used together, and this contstraint needs to be enforced.
2. **Singleton:**
   - **Intent:** Ensure a class only has one instance, and provide a global point of access to it.
   - **Applicability:** It is used when there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
3. **Composite:**
   - **Intent:** Compose objects into tree structure to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
   - **Applicablity:** It is used when clients need to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.
4. **Decorator:**
   - **Intent:** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
   - **Applicability:** It is used when extension by subclassing is impractical and to add responsibilities to individual objects dynamically and transparently, that is, with out affecting other objects.
5. **Template Method:**
   - **Intent:** Define the skelton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
   - **Applicability:** It is used when common behavior among subclasses should be factored and localized in a common class to avoid code duplication.

## B. PREDICATE-TEMPLATES

1. **package**(*PackageName*[4]):
   Clause package indicates that *PackageName* is a package. This is useful to reason about the classes and files across the packages.
2. **code**(*Class*,*FileInfo*):
   Clause code indicates that the class *Class* is part of the code in file *FileInfo*. The *FileInfo* variable also contains line number where the class *Class* is declared in the file. This is useful to extract code to refactor from the file mentioned in the *FileInfo* variable.
3. **field**(*DataMember*) :
   Clause field indicates that *DataMember* is a field of some class. The *DataMember* contains both field name and class or interface in which it is a member. This is useful to reason about the relationships such as association and aggregation.
4. **type**(*DataMember*, *Type*) :
   Clause type indicates that *DataMember* is of type *Type*. This is useful to reason about the objects and local variables of a method and parameters of a method, etc.
5. **return**(*Method*, *Type*) :
   Clause return indicates that method *Method* has return type *Type*. This is useful to reason about objects that are being returned.
6. **modifier**(*Member*, *AccessSpecifier*) :
   Clause modifier indicates that member *Member* has modifier *AccessSpecifier*. *AccessSpecifier* may be public, private, protected, default. This is useful to reason about members of the class or interface and inheritance type.
7. **method_has_switch**(*Method*, *Class*,*LineNumber*) :
   Clause method_has_switch indicates that method *Method* in class *Class* has a switch statement at line number *LineNumber*. This is useful to reason about nested conditional statements.
8. **creates**(*Method*, *Class*) :
   Clause creates indicates that the method *Method* creates an object of class *Class* through hard-coded constructor statements. This is useful to reason about object creation.
9. **parentswitch**(*StartLineNumber*, *EndLineNumber*, *N*) :
   Clause parentswitch indicates that switch statement starts from line *StartLineNumber* and ends at line *EndLineNumber* and it has *N* statements. If there exist nested switch statements then top most switch statement is called parent switch for all nested switch statements. Here, *StartLineNumber* is also used to identify(ID) unique parent switch statement. This is useful to reason about nested switch statements.
10. **case_block**(*StartLineNumber*, *EndLineNumber*, *ParentID*, *N*) :
    Clause case_block indicates that case block starts at line *StartLineNumber* and ends at line *EndLineNumber*. The case block is part of the switch statement whose ID is *ParentID* and it has *N* statements. This is also useful to reason about switch statements.
11. **case_var_value**(*Var*, *SwitchID*, *Val*) :
    Clause case_var_value indicates that switch statement *SwitchID* has a variable *Var* and each case block has a test value *Val* for this variable. This is useful to reason about switch statement test variables and their values.

[4]In this section and in Table 1, Prolog variables are denoted with italic font style.

12. **has_if_stmt**(*StartLineNumber*, *EndLineNumber*, *Method*, *Class*, *N*) :

    Clause has_if_stmt indicates that method *Method* of class *Class* has a if block which starts at line *StartLineNumber* and ends at line *EndLineNumber* and it has *N* statements. The *StartLineNumber* is also used as identification(ID) number for the if block. This is useful to reason about if and else conditional blocks.

13. **has_else_stmt**(*StartLineNumber*, *EndLineNumber*, *Method*, *Class*, *N*, *ParentIfID*):

    Clause has_else_stmt indicates that method *Method* of class *Class* has a else block which starts at line *StartLineNumber* and ends at line *EndLineNumber* and it has *N* statements. The else block is associated with if statement whose ID is *ParentIfID*. This is useful to reason about if and else conditional blocks.

# C. PROLOG RULE TO FIND DERIVED CLASSES

The rule that is shown in Table C.1 finds the list of derived classes in an inheritance tree whose base class is **Base**. The predicate **subtree_list** finds the immediate subtrees of class **Base**, these include the classes which **extends** the base class and the interfaces which **implements** the base interface. The number of elements in the subtree should be at least two. This means, there should exist at least two derived classes.

**Table C.1: Prolog Rule to Find Derived Classes in an Inheritance Tree**

```
derived_class_list( DerivedClassList, Base):-
        subtree_list(L),
        member(Tree,L),length(Tree,N),
        nth1(N,Tree,Base),
        setof(Derived,(member(Derived,Tree),
        not(Derived==Base)), DerivedClassList),
        not(DerivedClassList==[]).
```