Advances in Engineering Software 41 (2010) 519–526

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Design pattern recovery based on annotations

Ghulam Rasool *, Ilka Philippow, Patrick Mäder

Faculty of Computer Science and Automation, Software Systems/Process Informatics Group, Technical University of Ilmenau, Germany

ARTICLE INFO

Article history:
Received 30 May 2009
Received in revised form 4 September 2009
Accepted 15 October 2009
Available online 24 November 2009

Keywords:
Reverse engineering
Design patterns
Annotations
Program comprehension
Formal specifications

ABSTRACT

Design patterns have been widely used for developing flexible, extensible and perceptible applications to produce effective, reliable, verifiable and easily maintained software systems. The main advantage of using patterns is to take the edge of using best practices and experiences of others in solving the challenging tasks. Patterns have been extensively tested in different applications and reusing them yields the quality software. In this paper, we present a design pattern recovery approach based on annotations, regular expressions and database queries. We define the varying features of patterns and apply rules to match these features with the source code elements. Our novel approach reduces the search space and time for detecting patterns by using appropriate semantics of annotations from large legacy systems. We have tested our approach as proof of concept on motivating examples, and the obtained results are very encouraging.

1. Introduction

The reverse engineering process is highly influenced by the assumptions, hypothesis and abstractions to analyze large legacy applications due to limitation of automated tools. Tools are not capable and scalable to support the large systems that have different implementation styles. The most critical phase during reverse engineering process is the recovery of design information at different levels of abstraction. The detection of high-level abstractions, such as design patterns, in a software artifact represents an important step in program comprehension, refactoring, reengineering legacy applications and, hence, can be extremely useful during maintenance. The static as well as dynamic analysis approaches with manual, semi-automated and automated processes are used to recognize the design patterns from different legacy applications.

The static and dynamic pattern recovery approaches have their strengths and limitations. The static recovery approaches recognize pattern classes based on their inheritance, association, composition, instantiation and generalizations, etc., but the approaches fail to recognize some patterns whose structural signature is very weak or variable. The bridge pattern is cited as an example. The static analysis tools are not able to handle all types of associations and friend relationships in classes, which are important for pattern recovery [1]. Apart from these problems, the more serious constraints are the recovery of nonfunctional requirements (e.g. loose coupling between specific classes in a pattern). The dynamic analysis approaches cover areas such as performance optimization, software execution visualization, behavioral recovery and feature to code assignment. These approaches capture system behavior, but they are not practical in verifying the logic of a program. It complicates the search by expanding the set of candidate classes and results in analyzing more unrelated execution traces. The current pattern recovery approaches use different tools to get intermediate representations of source code (UML, AST, Parse trees, etc.) instead of directly using the source code. The choice of intermediate representation format directly affects the choice of the algorithms for discovery [2]. We believe that structural analysis should be used with low-level analysis of code and annotations to narrow down the search space of detecting patterns.

Several techniques have been used for recovery of design pattern instances from legacy source code. The recovery of different instants of design patterns from source code becomes arduous due to the following problems:

(1) Design patterns have different implementation variants, and there is no standard formal definition for each design pattern which is accepted by the whole research community.
(2) “There are several ways to implement the various relations in classes like delegation, aggregation, etc. which make the pattern recovery process difficult”.
(3) “Some programming languages provide library classes which facilitate pattern implementation but they complicate the recovery process”.
(4) “There is no benchmark system which can validate the results of different approaches”.
(5) The instances of different design patterns are scattered in the source code, and there are no formal rules for their composition.

* Corresponding author. Tel.: +49 17664822342.
E-mail address: Ghulam.rasool@tu-ilmenau.de (G. Rasool).

0965-9978/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.advengsoft.2009.10.014

(6) Pattern descriptions are abstract, informal and are usually not documented in the source code.
(7) Human opinion in pattern mining process is very desirable in some cases as it reduces the number of false positives.
(8) Most approaches are language specific and are not able to recover patterns from other languages.
(9) "Some approaches take the intermediate representation of the source code which affects the algorithms for pattern recovery".

Some of the above mentioned problems are also highlighted by [2,3] as quoted previously. These problems demonstrate many open issues in design pattern recovery and reflect the attention of reverse engineering community towards automated detection of design patterns. We do not claim to address all of the earlier mentioned issues in this paper, but we plan to overcome some of earlier mentioned obstacles with our pattern extraction approach. In this paper, we present our pattern detection approach which is different from other approaches on the basis of the following grounds:

(1) We present a novel, scalable and lightweight method for detection of varying features of design pattern which is based on annotations, database queries and regular expressions.
(2) Our approach suggests/request annotations which are used for forward engineering to document the patterns and for reverse engineering applications to detect the patterns.
(3) The approach is more efficient from the other methods because we reduce the search space and time by using appropriate semantics of annotations for detection of patterns.
(4) We take the advantage of using intermediate representation from Sparx Enterprise Architect Modeling tool (EA) [4] and use directly source code for extracting missing relationships by using regular expressions.
(5) Our approach is customizable because pattern features, rules, SQL queries and regular expressions are not hard coded in the source code.
(6) Our approach can be used to detect patterns from multiple languages supported by EA.
(7) Finally, we plan to detect the overlapping and composition of different instances of patterns.

There is still no agreed upon solution on design pattern recovery to date, because each approach takes its own definition for each design pattern. It causes the inconsistency in the results of different approaches used in the literature [5–8]. So, it is still premature to claim that design pattern recovery is fully automatic. It is a challenge for the reverse engineering and reengineering community to prove the automation of pattern recovery.

2. Related work

Design pattern recovery techniques have used structural analysis, behavioral analysis and semantic analysis or combination of "those" two/three to extract patterns from the source code. The review of some important attempts in the area of pattern recovery is given below:

The work presented in [9] used D^3 (D-cubed) tool for detection of singleton, factory method, abstract factory method, builder and visitor "pattern" from java source code. The approach establishes the program metamodel to formulate the definitions of design patterns in first-order logic. They translate the logic formulae to SQL and execute the queries against a database containing program metamodel which is manual, laborious and error prone activity. The approach has discovered some nonstandard variants of design patterns that are not detected by the FUJABA [10] and PINOT tools [11]. Our approach is different from their approaches because we use annotations, SQL queries, regular expressions and EA for pattern recognition. Their metamodel supports only Java language, but our approach can deal with multiple languages supported by EA. Secondly, we translate pattern features into SQL queries and regular expressions automatically.

Dong and Zhao [3] has presented an approach on the classification of design pattern traits. They describe different characteristics of design patterns as their traits in the form of predicates. Each predicate is classified into different groups and levels. Their pattern traits can be used by the other pattern matching tools. The characteristics of each pattern are semiformal and cannot be verified.

The approach presented in [6] used automatic pattern recovery technique and gets intermediate representation of the source code using Rational Rose. The capability of the approach depends on the extraction capability of the Rose tool. The technique has improved the precision and recall values on locally developed software applications that cannot assure the accuracy of the approach.

Dong et al. [2] have presented a review of different architecture and design pattern recovery techniques. They have discussed results obtained by different approaches and analyzed the potential reasons for the disparity of results. The authors suggest the benchmark system to validate the results obtained by different pattern recovery approaches.

Ref. [12] has presented an ontology-based architecture for pattern recognition. The approach uses parser, OWL ontologies and an analyzer. The approach integrates the knowledge representation field and static code analysis for pattern recognition while it lacks support for dynamic analysis, which is important for discovery of some patterns.

Meffert [13] has defined and used annotations as pattern templates that are used for refactoring the source code. The approach supports the design patterns only for program comprehension and documentation. The annotations used help maintenance programmers and clients of library code, but are not applied for design pattern detection.

The approaches presented in [14,15] convert their source code into Abstract Syntax Graph (ASG). They use different algorithms to identify candidate class structures based on graph transformation rules and deductive analysis. They are successful to detect some subset of patterns.

The approach presented in [16] has focused on the composition of design patterns. A concept of composition of patterns with respect to overlap is formally defined based on specialization and lifting operations on patterns. The approach is helpful for composition and formal verification of patterns but has no connection with the recovery of patterns, which is our major concern. We suggest that formal and informal semantics of annotation in the source code can support the pattern recovery process.

The work in [17] has proposed a design pattern recovery approach based on a visual language parsing technique. The approach first constructs an UML class diagram, which is represented in SVG format. They map the class diagram with the visual language grammar to detect different patterns. Silva et al. [18] have used a combination of formal methods and functional strategies to reverse engineer the graphical user interface of legacy applications. Their approach extracts only the abstract model of the user interface directly from GUI legacy source code. The approach is not generic, recovers only limited number of patterns and is limited only to Java programming language.

Ref. [19] has presented an approach for pattern detection using bit-vector algorithm. They express the problem of design pattern detection with different operations on a finite set of bit-vectors. They have performed experiments on JHotDraw, Juzzle and QuickUML with the abstract factory and the composite design patterns. However, the precision of their approach vary for different patterns. In [7], behavioral analysis is performed on Java.Awt package and JCL libraries. The sequence diagrams are used as rules for characteristics of different patterns. The approach has discovered only bridge, strategy and composite patterns successfully.

The tool PINOT presented in [11] uses static program analysis to extract program intent in order to recognize patterns. The tool claims to recover patterns by taking information from AST and obligatory definition of design patterns. The tool recovers all the GOF [20] patterns from the source code but with some false positives and false negatives.

3. Annotations for pattern recovery

The use of annotations for documenting legacy code is tacit to allow more ease instead of specific naming conventions and coding styles. The annotations have no direct effect on program semantics, but they are helpful for the analysis tools and libraries that are used for program comprehension and documentation. Annotations can be analyzed statically before and during compilation and can be read from source files, class files or reflectively at run time. Annotations contain intensions of developers that can be used to document the patterns and provide support for the recovery of design patterns from the source code. The pattern matching tools can use the formal semantics of annotations for analysis of source code. The Java Specification Request 305 [21] aims at introducing annotations for software defect detection. The developers can add the intentions related to various code elements through annotations.

Annotations add metadata information to different artifacts in the source code, and this information can be processed by different tools (compilers, javadoc, etc.). The existing mechanisms of using annotations have been adequate for general purpose such as documentation and refactoring of source code, but they do not support in complicated uses, and particular have a lack of support for design pattern recovery. The programmers use annotations for description of patterns that are not sufficient for pattern recovery. When one class is used in multiple patterns for different intentions, the existing method of annotations fails to recognize that class for the specific pattern. Due to missing semantics in the source code as well as in design pattern definitions, it is suggested to add semantic information in source code via annotations.

Annotations express design decisions that may be implicit, or described in documentation, but not easily available to tools for analysis. The pattern matching and analysis tools can use explicit semantic of annotations that are used in source code to facilitate the program comprehension, code transformations, pattern recovery and bug detection.

The annotations defined in [13] are used for selection of a particular block of the source code with special focus on refactoring the source code. We have defined annotations that are used for pattern documentation and recovery. The defined annotations could be used by machine as well as by human. The machine-configurable part can be used by the compiler for detecting errors or suppressing warnings. The human part is used for JavaDoc to maintain the documentation of the artifacts and changes made in the source code. We have defined more than 50 annotations that reflect the intentions of the designer in the source code. We are extending and modifying the list of annotations defined in [13] to detect the similarity of different annotations used in multiple patterns. Due to limitation of space, the following subset of annotations is shown as sample for maintaining documentation and recovery of design patterns:

(1) @abstract {notification |[interface, access_to_subsystem]|state_management|list_traversal|object_identity|handling},
(2) @compose {object} from {different_objects|related_objects},
(3) @decouple {receiver} from {sender},
(4) @object {instances} {share_by_introducing} {state_handlers|intrinsic_state|central_instance},
(5) @decouple {implementation} by {dynamic variation lists},
(6) @decouple {sender} from {receiver},
(7) @provide {handlers} for {requests|expressions},
(8) @ instantiation {eager|lazy|replaceable}.

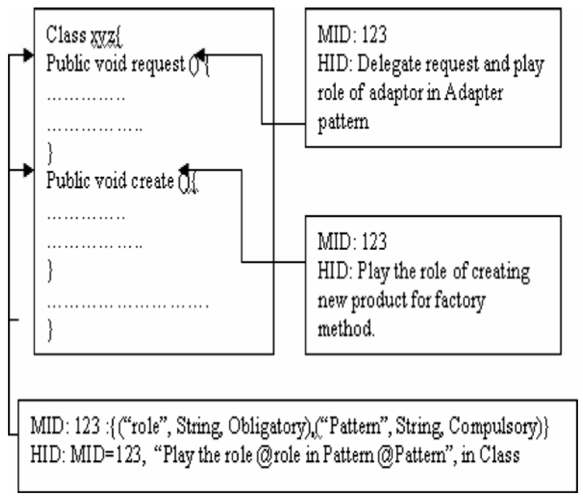
Table 1 shows the list of annotations that relates to specific design patterns. The developer can put these annotations in the source code as guidelines for the documentation and maintenance of legacy systems.

The annotations for patterns can also be specified by using formal specification languages. The specification for singleton and proxy patterns is shown in Fig. 2. These specifications are verified by using the Z/EVES [21] tool. These formal specifications of annotations may also be used as template for each design pattern. The Z specification can be converted into VDM, and tool support is available to convert it into Java source code.

The combination of a human as well as a machine-readable part for annotations is important for our pattern recovery approach. Our pattern detection process matches the similarity between the identification number for the annotation in the source code and in the annotation.type file. Our intention is to use the human-readable part of annotations for the static analysis of source code and detection of structural design patterns. The machine-configurable part will be used for dynamic analysis. The combination of human as well as machine semantics of annotations is very helpful for our pattern detection process. Fig. 1

Table 1
Group of annotations related to design patterns.

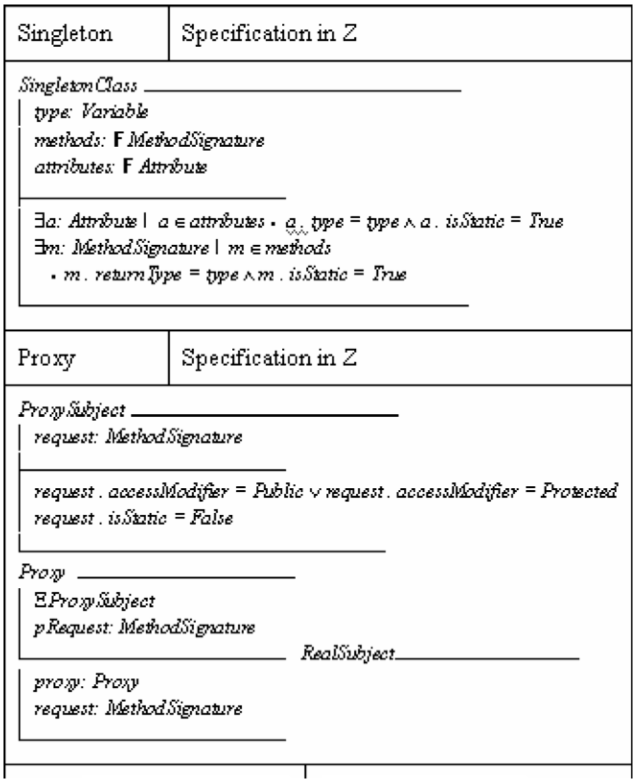
Pattern/annotation	1	2	3	4	5	6	7	8
Singleton		×	×	×				×
Composite		×		×		×		
Adapter	×							
Factory method			×		×			×
Proxy	×		×				×	


```
Class xyz{
Public void request (){
……………
………………
}
Public void create (){
……………
………………
}
…………………………
}
```
MID: 123
HID: Delegate request and play role of adaptor in Adapter pattern

MID: 123
HID: Play the role of creating new product for factory method.

MID: 123 :{("role", String, Obligatory),("Pattern", String, Compulsory)}
HID: MID=123, "Play the role @role in Pattern @Pattern", in Class

Fig. 1. Annotated class in source code with annotation. Type.



Singleton	Specification in Z

```
SingletonClass ____________________
| type: Variable
| methods: F MethodSignature
| attributes: F Attribute
|____________________
| ∃a: Attribute | a ∈ attributes • a . type = type ∧ a . isStatic = True
| ∃m: MethodSignature | m ∈ methods
|   • m . returnType = type ∧ m . isStatic = True
|____________________
```

Proxy	Specification in Z

```
ProxySubject ____________________
| request: MethodSignature
|____________________
| request . accessModifier = Public ∨ request . accessModifier = Protected
| request . isStatic = False
|____________________

Proxy ____________________
| ΞProxySubject
| pRequest: MethodSignature
|____________________  RealSubject____________________
| proxy: Proxy
| request: MethodSignature
|____________________
```

Fig. 2. Formal specifications for annotating patterns.

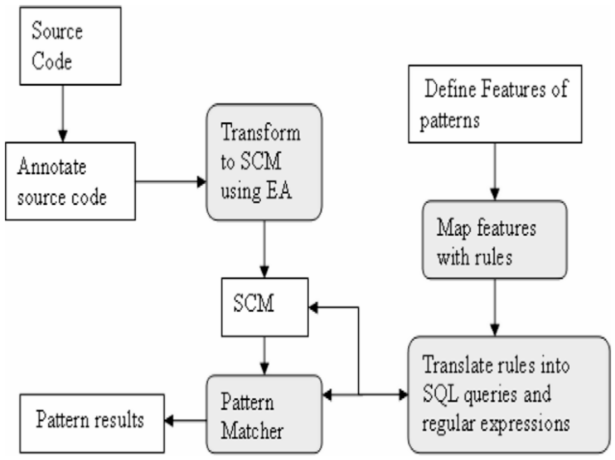

Source Code
Annotate source code
Transform to SCM using EA
SCM
Pattern Matcher
Pattern results
Define Features of patterns
Map features with rules
Translate rules into SQL queries and regular expressions

Fig. 3. Block diagram with stages of the pattern detection approach.

shows the application of a machine of an annotation and a human identification number (HID) for detecting Adapter and factory method pattern operations in the source code. The references of source code annotations are stored in an annotation.type file as shown in Fig. 1. Our pattern recognition process examines annotations from the source code and detects the candidate artifacts for pattern detection. In order to evaluate our method, we have partially annotated the source code manually to perform some experiments on different examples. We recommend the developers to use these annotations in the source code during development of applications that can facilitate the documentation and recovery of design patterns.

4. Pattern recovery approach

Our pattern recovery approach is based on regular expressions, SQL queries and annotations. We used regular expressions for extracting different relationships from the legacy applications which are missed by the EA tool. Regular expressions have a strong matching power and a simple syntax and are flexible enough to extend the pattern specifications. The lexical pattern matching tools like GREP [22] use regular expressions for pattern matching. The database queries are used for retrieval of data from relational database management systems. We used SQL queries to extract different relationship from the source code model extracted by the EA tool. The annotations that are used to describe the semantics of the code related with our pattern extraction approach as explained in Section 3.

The block diagram of our approach is shown in Fig. 3, which follows the following steps:

(1) We take the source code and put annotations on the source code manually as mentioned in Section 2.
(2) The annotated source code is reverse engineered using EA tool and we get the source code model including various relationships between different artifacts in the source code. The source code model does not contain information about delegation, aggregations and friend relationships between classes, which are important for pattern discovery.
(3) The varying features of each pattern are defined by using the definition of each pattern as defined by GOF [20]. The features are concatenated into rules to search the desired pattern.
(4) The Rules containing varying features are translated into SQL queries and regular expressions for searching the desired pattern.
(5) Due to the limitation in the reverse engineering capabilities of EA tool [2], we use regular expressions to extract delegation, aggregation, etc. relationships directly from the source code.
(6) We have developed a prototyping tool as Add-In with Visual Studio.Net framework to perform primary experiments on different motivating examples.

4.1. Implementation variants of design patterns

We have detected the following possible variations in structure of factory method in JHotDraw 6.1.2 [23] as shown in Table 2.

(1) The factory method class extends to its creator, and the factory method operations return the new concrete product which extends the product.
(2) The factory method class extends to its creator, and the factory method operation returns the concrete creator to its creator.
(3) The factory method class extends to its creator, and the factory method operation returns the factory method class which extends its creator.
(4) Parameterized factory methods may contain multiple product variants depends upon the selected parameter.

In proxy pattern, we search proxy class holding reference of real subject class as candidate elements for this pattern based on the annotation. The 17 instances of proxy are detected by our approach in JHotDraw.

Table 2
Various implementation styles of the factory method.

Struct1	Struct2	Struct3	Struct4	Total
81	6	3	0	90

The proxy pattern has following implementation variants.

(1) The proxy class extends the subject as proxy interface and has association with the real subject through its instance.
(2) The proxy class and real subject classes extend the subject class and proxy holds reference of real subject.
(3) Another variation of proxy is that only proxy classes hold reference of subject class which can also play the role of real subject class.

4.2. Pattern features and rules

We define the varying features of each pattern and map the features with rules to decide about the existence of pattern in the source code. Our pattern features definitions are similar to [3] with the addition of allowing features for the selection of patterns. We also define the rules for matching varying features of each pattern. Each feature is translated into a SQL query or regular expression according to the artifacts available in the source code model. The features for proxy pattern are shown in Table 3.

The features defined in Table 3 are mapped to rules to decide about the presence of pattern. So, the possible rules for proxy pattern will be as follows:

Rule 1: If (F1, F2, F3, F4, F6, F8) then proxy pattern is found.
Rule 2: If (F1, F2, F5, F7, F9) then proxy pattern is found.
Rule 3: If (F1, F2, F4, F6, F8) then proxy pattern is found.

The defined features can be reused to create and customized rules by adding new features or removing existing features in order to find other patterns or variations of a pattern. For example, the Adapter uses the same features as defined in Rule 1(Proxy Pattern) and the additional feature F10 to match the different structure of the Adapter pattern. The only parameter in F1, F2 and F4 will change. So, the Rule 4 is defined for selection of Adapter as given below.

F10: Has delegation (C1, M1, C3, and M2): It means that method M1 of the Adapter class delegate requests to method M2 of Adaptee class.

Rule 4: If (F1, F2, F3, F10) then Adapter pattern is found.

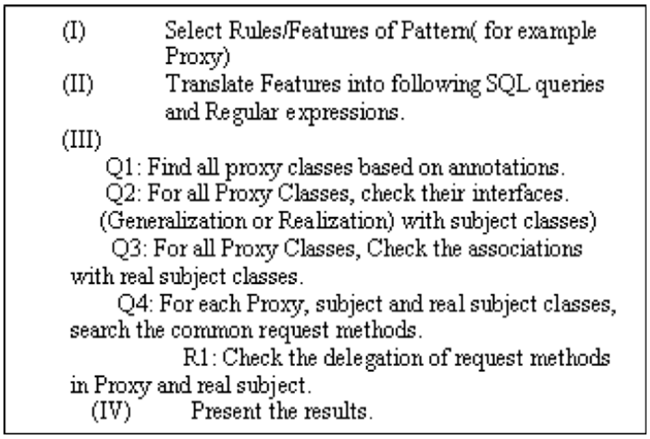
4.3. Algorithms for analysis

Different approaches use different algorithms for pattern recovery due to various implementation variants of the single design pattern. Most approaches get the intermediate representations from the source code which affects the algorithm for pattern recovery. We have analyzed the algorithm used in [6]. The algorithms are hard coded in the source code and fail to detect variants of patterns. Our approach takes the advantage of getting intermediate representations of source code through the Enterprise Architecture tool [5] into a source code model. The missing information in the intermediate model is extracted directly from source code using regular expression-based pattern definitions. The used annotations reduce the search space for detecting different patterns. For example, while extracting proxy pattern, we will extract all the proxy and real subject classes based on annotations and then explore the relationship between pair of proxy, real subject and subject classes.

Our pattern recognition approach uses a combination of annotations, SQL queries and regular expressions to match with varying features of design patterns. The SQL queries and regular expression patterns are customizable and not hard coded in the source. The sample pseudo code for proxy pattern detection is shown in Fig. 4. The variant of proxy can be matched just by selecting the desired features of the pattern as explained in Section 4.2.

4.4. Overlapping and composition

Overlapping is very important for reusability, because some classes are used in different patterns and play multiple roles. There exist different types of overlaps in design of different systems. For example, the creator class in the factory method pattern and the subject class in the proxy pattern can overlap. Similarly, during analysis of the Apache Ant 1.6.2 [24], it has been observed that some classes are working as interface for composing different patterns. For example, the Task.java is used in different patterns (Adapter class in Adapter pattern, component class in composite pattern, mediator and colleague class in mediator, etc.) as shown in Fig. 5. Similarly, Table 4 shows the overlapping of different classes in proxy and factory method patterns. Composition is important because different design patterns are used for plumbing the frameworks which are called architecture patterns. The architecture patterns are composed from different design patterns. For example, the model–view–controller (MVC) architecture pattern



```
(I)     Select Rules/Features of Pattern( for example
        Proxy)
(II)    Translate Features into following SQL queries
        and Regular expressions.
(III)
     Q1: Find all proxy classes based on annotations.
     Q2: For all Proxy Classes, check their interfaces.
    (Generalization or Realization) with subject classes)
      Q3: For all Proxy Classes, Check the associations
 with real subject classes.
       Q4: For each Proxy, subject and real subject classes,
 search the common request methods.
           R1: Check the delegation of request methods
 in Proxy and real subject.
   (IV)     Present the results.
```

Fig. 4. Pseudo code for proxy pattern detection.

Table 3
Various implementation styles of proxy pattern.

Features	Description
F1: Has stereotype (C1)	It means that class has stereotype C1 (proxy class)
F2: Has inheritance (C1, C2)	It means that proxy class inherits the C2 (subject class)
F3: Has inheritance (C3, C2)	It means that real subject class is inherited from C2 (subject class)
F4: Has association (C1, C3)	It means that proxy class maintains the reference of C3 (real subject class)
F5: Has association (C1, C2)	It means that proxy class maintains reference of subject class
F6: Has delegation (C1, M1, C3, M1)	It means that proxy class delegate request to real subject class
F8: Has common operation (C1, C2, C3 and M)	It means that proxy, subject and real subject classes have common request method M
F9: Has common operation (C1, C2, and M)	It means that proxy and subject classes have common request method M

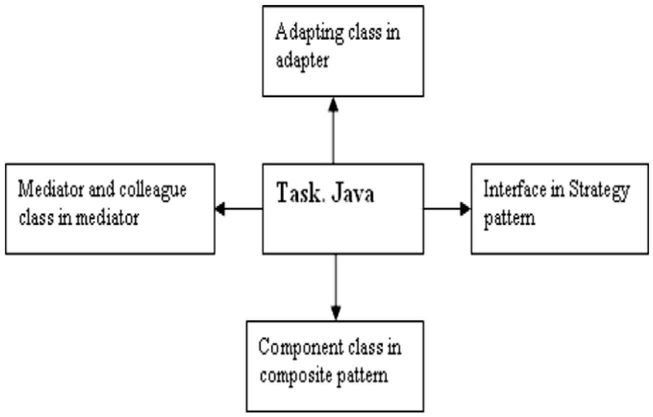

Adapting class in adapter
Mediator and colleague class in mediator
Task. Java
Interface in Strategy pattern
Component class in composite pattern

Fig. 5. Overlap operations in different patterns.

[27] uses instances of the observer pattern, the strategy pattern and the composite pattern. Currently, we are manually analyzing the composition of different patterns in open source systems to build a catalog of rules for detecting composition. The composition of different patterns can be used to extract the design and architectural model of the legacy applications.

5. Tools for design pattern recovery

Tools can be used for the pattern representation, recognition, combination, generation and application. The efficiency and completeness of a pattern matching tool is determined by matching different pattern implementation variants. Most of the design pattern tools have little or no support for documenting the presence and usage of patterns in the source code. Some tools are language dependent and require complete source code without syntax errors. The tools also differ in matching algorithms, pattern descriptions, pattern representations and precision. We have developed a prototyping tool as Add-In with Visual Studio.Net framework to perform some initial experiments on different motivating examples. The proposed architecture of our tool is shown in Fig. 6. The tool consists of data, code and presentation modules. The data module contains pattern rules, pattern features, SQL queries and regular expressions. The artifacts in this module are independent from other modules and are customizable. The code module consists of application programs written in C sharp with Enterprise Architecture as Add-In. Code module takes input from data module to execute the application programs. The presentation module is used to represent the recovered pattern results. The presentation of recovered pattern results play very important role in the program comprehension. Most of the pattern matching tools just give information about the presence of pattern in the source code and provide no idea about location of participating classes in the pattern and their multiple roles in the other patterns. We are currently working on the complete development of our prototyping tool. The code and data modules of our prototyping tool are working to extract different results, and we plan to extend the presentation module. The presentation module will be used to show the overlapping and the composition of different classes in multiple patterns.

We have already a custom built tool DRT [25] to extract different artifacts from the source code of legacy applications. DRT uses regular expressions for pattern matching and have good library of different regular expression patterns that we have already used in [26] for extraction of different artifacts. The pattern specifications of DRT are abstract and can be used to detect different artifacts from source code of multiple languages. We take the advantage of using regular expression patterns of DRT in our new prototyping tool, because Visual Studio.Net framework supports the parsing of regular expressions. The library of regular expression patterns is stored in a text file, and it is used by our prototyping tool. These pattern definitions can be iteratively used and may be refined according to the requirement of matching varying features of different patterns. So, the combination of regular expressions, SQL queries and semantics of annotations make our tool unique for fast pattern extraction.

The proposed architecture of our new prototyping tool is shown in Fig. 6. We have performed some experiments on different patterns and compared the results with the other tools as shown in Table 5. However, we have performed experiments on the subset of open source systems like JHotDraw 6.1.2 and Apache Ant 1.6.2. Our primary results are very promising and encouraging.

6. Experimental examples

We have used our custom-build tool DRT and Add-In created from [4] for performing experiments on some motivating examples. For example, the singleton pattern is extracted using the method described in Section 4. The singleton classes are extracted from source code using annotations. We used combinations of SQL queries and regular expressions to match different features of

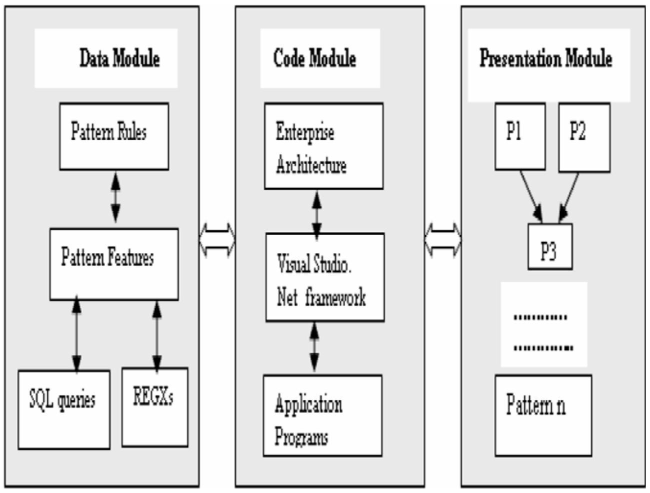


Fig. 6. Proposed architecture of prototyping tool.

Table 4
Overlapping in different pattern classes.

Class	Factory method	Proxy
File location: src/org/jhotdraw/samples/pert/PertFigure.java	PertFigure is CFM class	PertFigure is a proxy
File location: src/org/jhotdraw/figures/AttributeFigure.java	AttributeFigure is FM	AttributeFigure is a proxy interface
File location: src/org/jhotdraw/samples/javadraw/JavaDrawApplet.java	JavaDrawApplet is CFM	JavaDrawApplet is a proxy
File location: src/org/jhotdraw/standard/AbstractFigure.java	AbstractFigure is FM.	AbstractFigure is a proxy interface
File location: src/org/jhotdraw/figures/RectangleFigure.java	RectangleFigure is CFM/FM	RectangleFigure is a proxy
File location: src/org/jhotdraw/util/collections/jdk11/ListWrapper.java	ListWrapper is CFM	ListWrapper is a proxy

CFM: concrete factory method; FM: factory method.

Table 5
Comparison of tools.

	PINOT	FUJABA	Our approach
Singleton	++	+	++
Factory method	+	+	++
Adapter	++	+	++
Composite	+	+	+
Proxy	++	–	++
Observer	++	+	++
Visitor	++	–	+

–: Tool has no support in detecting pattern.
+: Tool detects patterns but with high rate of false positive and false negative.
++: Tool detects patterns with very low rate of false positive and false negative.

singleton. The features of each design pattern are formally specified, and set of rules are defined that will match to each feature. Singleton classes are detected from source code based on annotations used in the source code. Secondly, we use different regular expression patterns and SQL queries to match the structure of singleton class. For example, the following SQL statement is used to detect the structure of singleton classes:

SQL: "Select t_object.Object_ID,t_object.Name, t_object.Scope, t_operation.Name,t_operation.scope,t_operation.Type,t_attribute.Name,t_attribute.Scope,t_attribute.Type,t_attribute.IsStatic from t_object, t_operation, t_attribute where t_object.stereotype = 'singleton' and t_object.Object_ID = t_operation.Object_ID and t_object.Object_ID = t_attribute.Object_ID and (t_object.name = t_operation.name or t_object.name = t_operation.Type) and t_attribute.type = t_object.Name and t_attribute.IsStatic = 1 and t_attribute.scope = 'private'"

The above SQL statement is used only to detect the static features of the singleton class. The FUJABA [10] uses only the static features of singleton pattern to detect the singleton instances, and many false positive are reported by it. The dynamic features of some patterns are important with static analysis to recognize the correct pattern instances. The behavior of the get instance singleton method, lazy instantiation, eager instantiation and subclass singleton instances are detected by using regular expression pattern matching. The appropriate semantics of annotations in the source code play key role to detect the dynamic features of the patterns. Similarly, we have recovered different variants of proxy pattern following pseudo code explained in Section 4. The user inputs only the features of proxy and multiple SQL queries, and regular expressions are generated to detect different instances of proxy. SQL queries used the results of previous queries to match with the features of different patterns. For example, the following SQL queries are generated to detect variant of proxy pattern as mentioned in Section 4.2 (Rule 1).

Query 1: "select object_id from t_object where stereotype = '%P1%'".
Query 2: "select end_object_id from t_connector where (connector_type = 'Generalization' or connector_type = 'Realisation') and (start_object_id = %PR0%"+")"
Query 3: "select end_object_id from t_connector where (connector_type = 'Association') and (start_object_id = %PR0%"+")"
Query 4: "select F.object_id, S.object_id, T.object_id, F.name from t_operation F, t_operation S, t_operation T where F.object_id = %PR0% and S.object_id = %PR1% and T.object_id = %PR2% and F.name = S.name and F.name = T.name and F.object_id<>S.object_id and F.object_id<>T.object_id".

The P1, PR0, PR1, PR2 are placeholders that store the results of different queries. The P1 stores by default the result of first query which is used to detect the stereotype of the artifacts used for pattern recovery. These results are used by other queries to match with different features of patterns. The delegation feature of proxy is extracted by using regular expression pattern matching.

Currently, we are performing experiments on different other patterns. We cannot completely compare the precision and recall with the other approaches, because we have performed experiments on subsets of open source applications by using annotations. We are in the phase of annotating the complete open source systems like JHotDraw 6.1.2 [23] and Apache Ant 1.6.2 [24] to compare accuracy of our approach.

7. Conclusions

The recognition of design patterns provides additional information related to the rationale behind the design. The semantics of annotations are very helpful for design pattern documentation and their recognition. We believe that appropriate semantic of annotations in the source code can gain significant benefits in program comprehension and are very helpful during the pattern recovery .The annotations used in our approach reduce the search space for detecting patterns from legacy applications. The designer can use these annotations as guidelines for the developers and the maintainers. The concept of our approach is novel, and initial results are very encouraging. We are currently working on the more extensive evaluation of our approach. Our lightweight approach takes advantage of information extracted by the source code model using EA tool as well as uses directly source code files to extract missing information in the source code model. The approach is also customizable because the SQL queries, pattern features, rules and regular expression patterns are independent from source code and can be changed according to the varying features of the patterns. We have evaluated our detection technique by adapting into various programs and comparing the results with other pattern detection tools as shown in Table 5. However, our detection results are purely based on annotations in the source code. Finally, we will extend our approach to detect different overlapping and composition of patterns that are analyzed manually in different open source systems.

References

[1] Koschke Rainer. What architects should know about reverse engineering and reengineering. In: Proceedings of the 5th working IEEE/IFIP conference on software architecture; 2005. p. 6–10.
[2] Dong Jing, Zhao Yajing, Peng Tu. A review of design pattern mining techniques. Int J Softw Eng Knowl Eng (IJSEKE) 2009;19(6):823–55.
[3] Dong Jing, Zhao Yajing. Classification of design pattern traits. In: Proceedings of 19th international conference on software engineering and knowledge (SEKE), Bostan, USA; July 2007. p. 473–6.
[4] Sparx System Architect Modeling tool. <http://www.sparxsystems.com/products/ea/> [accessed 05.05.09].
[5] De Lucia Andrea, Deufemia Vincenzo, Gravino Carmine, Risi Michele. A two phase approach to design pattern recovery. In: Proceedings of 11th European conference on software maintenance and reengineering (CSMR'07), Amsterdam, Netherlands; 21–23 March 2007. p. 297–306.
[6] Philippow Ilka, Streitferdt Detlef, Riebisch Matthias, Naumann Sebastian. An approach for reverse engineering of design patterns, vol. 4 (1). Springer-Verlag; 2004. p. 55–70.
[7] Wendehals L. Improving design pattern instance recognition by dynamic analysis. In: Proceedings of the ICSE workshop on dynamic analysis (WODA); May 2003. p. 29–32.
[8] Smith JM, Stotts D. SPQR: flexible automated design pattern extraction from source code. In: Proceedings of the ASE, IEEE Computer Society Press, Canada; 6–10 October 2003. p. 215–24.
[9] Stencel Krzysztof, Wegrzynowicz Patrycja. Detection of diverse design pattern variants. In: 15th Asia-Pacific Software Engineering Conference, Beijing, China; 3–5 December 2008. p. 25–32.
[10] <http://wwwcs.uni-paderborn.de/cs/fujaba/> [accessed 8.08.09].
[11] Shi N, Olsson RA. Reverse engineering of design patterns from java source code. In: Proceedings of the 21st IEEE international conference on automated software engineering (ASE'06), vol. 00; 2006. p. 123–34.

[12] Kirasic Damir, Bash Danko. Ontology-based design pattern recognition. Lecture notes in computer science 5177/2008; 20 September 2008. p. 384–93.
[13] Meffert Klaus. Supporting design patterns with annotations. In: Proceedings of the 13th annual IEEE international symposium and workshop on engineering of computer based systems (ECBS'06), Postdam, Germany; 27–30 March 2006. p. 445–51.
[14] Balanyi Z, Ferenc R. Mining design patterns from c++ source code. In: International conference on software maintenance (ICSM'03), Amsterdam, The Netherlands; 2003. p. 305–14.
[15] Niere J, Shafer W, Wadsack JP, Wendehals L, Walsh J. Towards pattern-based design recovery. In: Proceedings of the international conference on software engineering (ICSE'02), Orlando, FL, USA; 2002. p. 338–48.
[16] Bayley I. Zhu Hong. On the composition of design patterns. In: Proceedings of the 8th international conference on quality software, Oxford, UK; 12–13 August 2008. p. 27–36.
[17] Costagliola Gennaro, De Lucia Andrea, Deufemia Vincenzo, Gravino Carmine, Risi Michele. Design pattern recovery by visual language parsing. In: Proceedings of the 9th European conference on software maintenance and reengineering (CSMR'05), Manchester, UK; March 2005. p. 102–11.
[18] Silva JC, Campos JC, Saraiva J. Combining formal methods and functional strategies regarding the reverse engineering of interactive applications, vol. 4323. Berlin Heidelberg: Springer-Verlag; 2007. p. 137–50. ISBN:978-3-540-69553-0.
[19] Kaczor O. Gueheneuc Y-G, Hamel S. Efficient identification of design patterns with bit-vector algorithm. In: Proceedings of the 10th European conference on software maintenance and reengineering, Bari, Italy; 22–24 March 2006. p. 184–93.
[20] Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: elements of reusable object oriented software. Reading, MA: Addison-Wesley Publishing Company; 1995.
[21] Hovemever David, Paugh William. Status report on JSR-305: annotations for software defect detection. In: Conference on object oriented programming systems languages and applications, Montreal, Quebec, Canada; 2007. p.799–800.
[22] Abou-Assalaeh Tony, Ai Wei. Survey of global regular expression print (GREP) tools; 02 March 2004.
[23] JHotDraw StartPage. <http://www.jhotdraw.org>. [accessed 15.06.09].
[24] Apache Ant 1.6.2. <http://www.linuxfromscratch.org/blfs/view/6.1/general/> [accessed 24.07.09].
[25] Rasool G, Asif N. DRT tool. Int J Softw Eng 2007;1(1):67–71.
[26] Rasool G, Asif N. Software artifacts recovery using abstract regular expressions. In: Proceedings of the 11th IEEE multitopic conference, Comsats Institute of IT Lahore Campus, Pakistan; 28–30 December 2007. p. 1–6.
[27] Simon Beloglavec, Marjan Heričko. A composite design-pattern identification technique. J Inform (03505596) 2005;29(4):469–76.