Dynamic Analysis and Design Pattern Detection in Java Programs

Lei Hu and Kamran Sartipi
Dept. Computing and Software, McMaster University, Hamilton, ON, L8S 4K1, Canada
{*hu14, sartipi*}*@mcmaster.ca*

Abstract

Identifying design patterns within an existing software system can support understandability and reuse of the system's core functionality. In this context, incorporating behavioral features into the design pattern recovery would enhance the scalability of the process. The main advantage of the new approach in this paper over the existing approaches is incorporating dynamic analysis and feature localization in source code. This allows us to perform a goal-driven design pattern detection and focus ourselves on patterns that implement specific software functionality, as opposed to conducting a general pattern detection which is susceptible to high complexity problem. Using a new pattern description language and a matching process we identify the instances of these patterns within the obtained classes and interactions. We use a two-phase matching process: i) an approximate matching of class attributes generates a list of candidate patterns; and ii) a structural matching of classes identifies exact matched patterns. One target application domain can be software product line which emphasizes on reusing core software artifacts to construct a reference architecture for several similar products. Finally, we present the result of a case study.

KEYWORDS: Dynamic Analysis; Design Pattern Detection; Feature-specific Scenario; Pattern Matching; Software Family; Data Mining.

1. Introduction

Software companies that satisfy the needs of a specific market segment develop products that share common sets of features [8]. These products are usually developed based on a reference architecture which consists of common parts and variable parts, where the variable parts can be modified to satisfy the evolving requirements of the new products. In this context, the evolutionary development of a software system starts from identifying the important features contained in similar products as well as identifying the reusable components based on the reference architecture [7].

In this paper, we propose a new approach based on a hybrid dynamic and static analysis to address the problem of reusing existing system's design patterns that correspond to specific software behavior as the goals of the recovery process. In this context, design patterns (i.e., common solutions to recurring design problems [11]) can assist a software engineer in comprehending and reusing design decisions and solutions adopted by the original software designers. Consequently, these patterns can be used in developing a family of similar systems that share the same core features.

The proposed framework identifies the existing design patterns in the key software features through two major parts: *dynamic analysis* and *pattern detection*. In dynamic analysis, we identify a group of key features of the subject system and generate a set of relevant task scenarios for each feature, namely feature-specific scenario set. Through scenario execution, pattern mining, and concept lattice analysis we obtain the classes that contribute in generating those features without any prior knowledge about the system. The obtained classes will form a search space to conduct the pattern detection process, where the design patterns are specified using a new pattern description language (PDL) that drives the pattern matching process. A pattern repository holds the specification of a number of design patterns. The pattern matching process recovers the instances of the design patterns in the repository in two phases: i) an approximate matching process generates a list of potential pattern instances for each target pattern, by comparing the number of class attributes in the search space; and ii) a structural matching compares the complete class structure of the target pattern against the structure of the candidate instance pattern.

In order to extract the core functionality of the existing systems the software solution providers need a set of diverse reverse engineering tools to be used for different projects and at different application domains [6]. The approach proposed in this paper contributes in such problem domain by the followings:

i) mapping software behavior to source code as a means to identify core classes that implement the key features of a software system; hence providing a reduced search

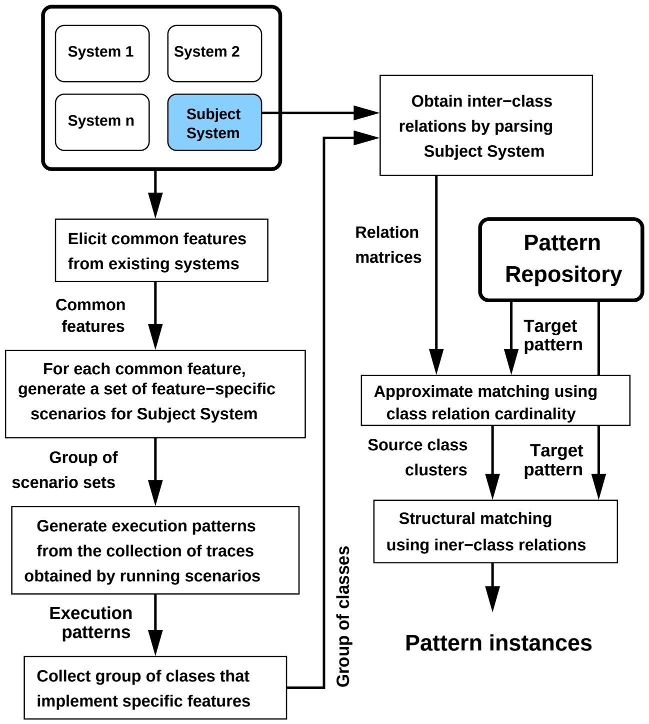

System 1
System 2
System n
Subject System
Obtain inter−class relations by parsing Subject System
Elicit common features from existing systems
Relation matrices
Pattern Repository
Common features
Target pattern
For each common feature, generate a set of feature−specific scenarios for Subject System
Approximate matching using class relation cardinality
Group of scenario sets
Source class clusters
Target pattern
Generate execution patterns from the collection of traces obtained by running scenarios
Structural matching using iner−class relations
Group of classes
Execution patterns
Pattern instances
Collect group of clases that implement specific features

Figure 1. The proposed framework for dynamic analysis and design pattern recovery.

space for design pattern recovery; and

ii) presenting a novel two-phase search technique and a pattern definition language to perform design pattern recovery.

2. Related work

In this section, we discuss relevant approaches in dynamic analysis and design pattern detection to our work.

In dynamic analysis of software systems, El-Ramly et al. [10] applied a sequential pattern mining technique to identify interaction patterns between graphical user interface components. Zaidman et al. [16] applied a web-mining technique on program dynamic call graphs, where nodes represent classes and edges represent method invocation. Eisenbarth et al. [9] proposed a formal concept lattice analysis to locate computational units that implement a certain feature of the software system. In contrast to the above techniques, our approach exploits a novel analysis technique to handle large sizes of the execution traces, and allows an intuitive and promising process of feature to component allocation.

We classify approaches to design pattern recovery (focus of this paper) into two major categories, as follows.

Structure-based pattern detection. In this category, the detection process identifies pattern instances that have the same pattern class structure as a target pattern. Nija Shi et al. [14] propose an approach to discover the GoF patterns from Java source code based on data-flow analysis on abstract syntax tree in terms of basic blocks. Lucia et al. [12] propose a two-phase approach to recover structural design patterns, where in the first phase the number of candidate patterns are reduced through analysis of class diagram structure, and in the second phase the real patterns are identified by user inspection.

Matrix-based pattern detection. In this category, the approaches store the inter-class relations in the software system as well as the target design patterns into different matrices. Thus, the pattern matching process is accomplished by matrix matching. Nikolaos et al. [15] present an automatic approach which uses a similarity score algorithm to detect design patterns. The design pattern detection is accomplished by calculating the similarity score between the matrices of system and those of target design patterns.

3. Proposed framework

Figure 1 illustrates the proposed approach for design pattern recovery. The framework consist of *dynamic analysis* to assign system features onto a set of system classes; and *pattern detection* to locate the instances of individual patterns in the software system, by comparing the target patterns in the pattern repository with software's class structure.

Dynamic analysis. The proposed dynamic analysis operates on the run-time execution traces of a set of subject features to locate the corresponding low-level system components that implement each feature. This process consists of the following steps: i) feature-specific scenario set generation; ii) execution traces generation; iii) execution pattern extraction from execution traces; and iv) execution pattern analysis.

Pattern detection. The proposed design pattern detection process consists of two phases *approximate matching* and *structural matching*. In the approximate matching phase, through identifying the eligible candidates for the main-seed class of the target design pattern, we reduce the search space for a target design pattern to a list of source-class clusters, each of which contains a candidate main-seed class. In the structural matching phase, we identify the structurally matched design pattern instances within the list of source-class clusters. The detail description of these two phases are discussed in Sections 4 and 5.

4. Dynamic analysis

We propose a dynamic analysis technique to locate the source code implementation of key features in object-oriented systems, which is an enhancement of the previous work presented in [13]. In the remaining of this section we will give a description for the process of dynamic analysis.

4.1. Execution pattern extraction

Scenario selection. According to the knowledge about the application domain, available documents, and user's guide of the subject system, we generate a set of relevant task scenarios where all scenarios share a specific software feature. We call this set of scenarios as feature-specific scenario set.

Execution trace generation. In this step, we use Eclipse Test and Performance Tools Platform (TPTP) [2] to instrument and collect execution information from the software system. By running scenarios of the feature-specific scenario set on the instrumented software system, we obtain the execution traces of each scenario in the form of entry/exit listings of the object invocations.

Execution pattern generation. By applying a sequential pattern mining algorithm on the execution traces of the specified feature, we can obtain the execution patterns of the feature. Here we use a modified version of the sequential pattern mining algorithm by Agrawal [4].

4.2. Execution pattern analysis

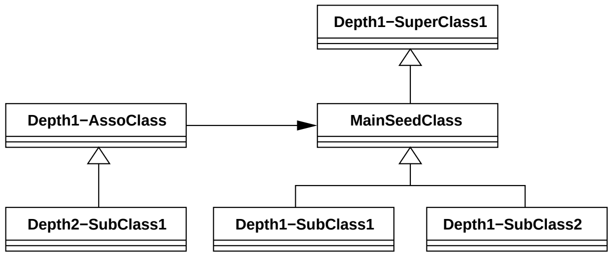
After obtaining the execution patterns of several specific features, we use concept lattice analysis to cluster the group of classes in the execution patterns that exclusively correspond to each specific feature; as well as the class clusters that are common to every scenario set. In this context, the clusters of common classes appear in the upper region of the lattice, and the clusters of feature-specific classes appear at nodes in the lower region of the lattice. Thus, a mapping between the software feature and its implementation is obtained at the bottom of concept lattice.

5. Design pattern detection

To avoid the combinatorial explosion in pattern detection process, we present a two-phase and semi-automated design pattern detection process where each design pattern is populated around a main-seed class.

5.1. Pattern description

Formally, a design pattern p can be represented as a tuple $< \mathcal{C}, \mathcal{R} >$, where $\mathcal{C}$ is a set of *pattern-classes* $\{c_1, ..., c_k\}$ and $\mathcal{R}$ is a set of *inter-class relations* among these pattern-classes. For two pattern-classes c_i and c_j in $\mathcal{C}$, $ShortestPath(c_i, c_j)$ returns the minimum number of inter-class relations traversed from c_i to reach c_j, regardless of the type of the inter-class relations [5]. The *Degree* of a pattern-class c_i in $\mathcal{C}$, denoted as $deg(c_i)$, is the number of the direct inter-class relations between c_i and all the other pattern-classes in the design pattern p.



```
Begin-PDL
  Pattern : $TargetPattern$
    Main-seed class : $MainSeedClass$
      Depth1 :
        Inherit_From :
          Depth1-SuperClass1
        Inherited_By :
          Depth1-SubClass1;
          Depth1-SubClass2
       in_Association :
         Depth1-AssoClass
     Depth2 :
       Seed-Depth1: Depth1-AssoClass
          Inherited_By :
            Depth2-SubClass1
     AbstractClasses :
       Depth1-SuperClass1
   End-Pattern
 End-PDL
```

Figure 2. Class diagram and PDL description of a target pattern.

Main-seed class. We observe that for each design pattern presented in [11], there exists at least one pattern-class which can reach any other pattern-classes in the design pattern within a shortest path value 2. We refer to this kind of pattern-class as *potential main-seed class*, whose formal definition is given below.

Potential main-seed class. In a design pattern $p = \langle \mathcal{C}, \mathcal{R} \rangle$, a potential main-seed class, denoted as c^{pms}, is a pattern-class $c^{pms} \in \mathcal{C}$ such that $\forall c_i \in \mathcal{C} \bullet ShortestPath(c^{pms}, c_i) \leq 2$. C^{pms} is referred to the set of all the potential main-seed classes in the design pattern p.

We propose a Pattern Description Language (PDL) to describe a generic pattern. PDL provides a convenient way to describe a design pattern in a precise way and allows the user to define any other pattern they desire to discover. Figure 2 presents the class diagram of a target pattern and its corresponding PDL description.

5.2. Pattern detection

The pattern detection consists of a two-step matching process, as: *approximate matching* to generate a ranked list of eligible candidate instance patterns; and *structural matching* to identify the structurally matched instance patterns within the ranked list of instances.

Approximate matching. In approximate matching, the main goal is to reduce the search space to a number of instance patterns that are sufficiently close to the target pattern. In this context, we specify a set of attributes for the main-seed of the patterns (both target pattern and instance patterns) whose values are used to compare these two patterns. Hence, we can rank eligible instance patterns in the search space and generate a short list of approximately similar instance patterns to the target pattern. The main-seed attributes include the number of *Inherit_From*, *Inherited_By*, *Association* and *Abstract* relations.

As shown in Figure 2, the main-seed class "$MainSeedClass$" possesses one *Inherit_From* relation, one *Association* relation, and two *Inherited_By* relations. Considering a search space as a set of classes $SP = \{c_1, c_2, ..., c_n\}$, for each class $c_i \in SP$ we define an attribute vector $Attr(c_i) = [a_1, ..., a_k]$ with cardinality k. Given the main-seeds c_t of the target pattern and c_i of the instance pattern, the approximate similarity function sim_{apx} is defined as:

$$sim_{apx}(Attr(c_i), Attr(c_t)) =$$

$$\begin{cases} \Delta(Attr(c_i), Attr(c_t)) & Attr(c_i) \geq Attr(c_t) \\ 0 & Else \end{cases}$$

$$\Delta(Attr(c_i), Attr(c_t)) = 1 - \frac{\sum_{j=1}^{k}(Attr_j(c_i) - Attr_j(c_t))}{\sum_{j=1}^{k} Attr_j(c_i)}$$

where $Attr(c_i) \geq Attr(c_t)$ means that the value of each element in the attribute vector $Attr(c_i)$ is greater than or equal to that of attribute vector $Attr(c_t)$. In this context, function sim_{apx} computes the approximate similarity value between the target pattern (represented by the main-seed class c_t) and the candidate instance pattern (represented by main-seed class c_i).

Algorithm "ApproximateMatching" receives the search space, class relation matrices, target pattern, and a cut-off threshold similarity value, and returns the list of eligible candidate instance patterns. The algorithm utilizes the function "$ComputeAttrValue()$" to compute the attribute values of a main-seed using the class relation matrices; and function "$GeneratePattern()$" to compose an instance pattern with two level classes using every class c_i (in different iterations) from the search space.

Systems	Version	# Classes	#Files	#LOC
JHotDraw	5.1	172	144	8419
JHotDraw	6.0b1	405	289	21091
JHotDraw	7.0.7	331	309	32122

Table 1. Statistics for the subject systems.

Structural matching. Structural matching algorithm deals with the identification of all the instances of the target pattern within a candidate instance pattern[1] obtained from the aforementioned approximate matching. The algorithm receives a candidate instance pattern, target pattern, and the class relation matrices. It returns one or more identified instance patterns within the candidate instance pattern. The algorithm utilizes the functions $GetDepth1Classes()$ and $GetDept2Classes()$ to retrieve the corresponding depth1 and depth2 classes from the PDL representation of the target pattern.

After a pattern instance is detected, a further user-assisted verification has to be performed to check whether the detected pattern is actually implemented within the subject software system or not. This verification is performed through browsing the source code and consulting with the existing design documents.

6. Case study

In this section, we discuss the results of applying the proposed approach on a Java open-source project, *JHotDraw* [1]. JHotDraw is a Java GUI framework which is used to draw two-dimensional graphics and it contains many instances of design patterns in its implementation.

Based on discussion in Section 1, we apply our proposed approach on three versions of JHotDraw, ver5.1, ver6.0b1 and ver7.0.7 to extract reusable software artifacts. The experiments are performed on a Windows XP professional edition running on a PC with a 1.5GHZ centrino processor, 512M bytes memory and 1G bytes virtual memory.

Table 1 presents several system statistics from three versions of JHotDraw systems in the case study. Because of space limitation, the results of execution trace extraction and execution pattern mining for 10 features of the three versions of JHotDraw systems are not presented in this paper, however similar experimentation can be found in our previous work [13]. In a further step, we supply the resulting execution patterns obtained from the sequential pattern mining to a concept lattice generation tool, ConExp [3]. Finally, we generate a search space by collecting all classes of the feature-specific concepts and augmenting this space by adding two levels of immediately related classes.

[1]Note that a target pattern usually has fewer classes than the candidate instance pattern, hence it may match with more than one sub-pattern instances within the candidate pattern.

	Rectangle	Round Rectangle	Ellipse	Polygon	Line	Move	Delete	Group	LineConnection	Text
Adapter	0/0/0	1/1/1	0/0/0	2/1/1	1/1/1	0/0/0	0/1/0	1/0/1	2/2/2	0/1/0
Observer	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	1/1/0	0/0/0	2/2/0	0/0/0	0/0/0
Proxy	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Decorator	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Strategy & State	1/1/1	1/1/1	1/1/1	1/1/0	1/1/0	1/1/1	1/2/0	1/0/1	5/4/3	1/2/1

Legend: A / B / C
A: data of JHotDraw 5.1, B: data of JHotDraw 6.0b1, C: data of JHotDraw 7.0.7

Table 2. Results of mapping between target patterns and 10 features in three versions of JHotDraw.

In the following phase of the experimental study, we apply pattern detection algorithms “ApproximateMatching()” and “StructuralMatching()” (discussed in Section 5) on the search space to detect all the pattern instances of the target patterns in the pattern repository. We describe structural information of each pattern using the proposed pattern definition language (PDL) and store it into the pattern repository. Currently, our pattern repository contains the following patterns: *Adapter, Proxy, Observer, Decorator, Bridge* and *Strategy & State*. To filter out the false-positive patterns in the detected pattern instances, we perform a manual verification on these resulting pattern instances by inspecting the corresponding source code. To correlate a detected pattern instance to a software feature, we check the overlap between the highly related classes of the feature (obtained from concept lattice) with the participating classes of the pattern instance. If there is an overlap, it means that there exists a relation between the feature and the pattern instance. Table 2 presents the correlation of detected pattern instances to the 10 features of the three versions of JHotDraw systems. The value in each entry of the table represents the number of the pattern instances that are correlated with the corresponding feature.

7. Conclusions

In this paper, we presented a two-phase approach to identify individual design patterns within a subject software system as a means to assist the construction of a reference architecture for a family of software systems, or for different versions of the same system. The main advantage of our approach over the existing approaches is incorporating dynamic analysis and feature localization in source code. This allows us to perform a goal-driven design pattern detection and focus ourselves on design patterns that implement specific software functionality as opposed to conducting a general pattern detection which is susceptible to high complexity problem. We have successfully experimented with JHotDraw system which is considered as a benchmark for design pattern recovery.

References

[1] Jhotdraw start page. http://www.jhotdraw.org, 2006.

[2] The eclipse test and performance tools platform, 2006. http://www.eclipse.org/tptp.

[3] Formal concept analysis toolkit version 1.0.1. http://sourceforge.net/projects/conexp.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE ’95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14, 1995. IEEE Computer Society.

[5] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *IWPC ’98*, pages 153–160. IEEE Computer Society Press, June 1998.

[6] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. Pulse: a methodology to develop software product lines. In *Proceedings of SSR ’99*, pages 122–131, New York, NY, USA, 1999. ACM Press.

[7] J. Bayer, J.-F. Girard, M. Wurthner, J.-M. DeBaud, and M. Apel. Transitioning legacy assets to a product line architecture. In *Proceedings of ESEC/FSE-7*, pages 446–463, London, UK, 1999. Springer-Verlag.

[8] P. Clements and L. Northrop. A framework for software product line practice. Technical report, www.sei.cmu.edu/productlines/framework.html, 2004.

[9] T. Eisenbarth, R. Koschke, and D. Simon. Derivation of feature component maps by means of concept analysis. In *Proceedings of IEEE CSMR’01*, pages 176–179, March 2001.

[10] M. El-Ramly, E. Stroulia, and P. Sorenson. Recovering software requirements from system-user interaction traces. In *SEKE ’02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 447–454, New York, NY, USA, 2002. ACM Press.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[12] A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi. A two phase approach to design pattern recovery. In *in Proceedings of CSMR07*, pages 297–306, Amsterdam, Netherlands, 2007. IEEE CS Press.

[13] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *ICPC ’06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC’06)*, pages 84–88, Athens, Greece, 2006. IEEE Computer Society.

[14] N. Shi and R. A. Olsson. Reverse engineering of design patterns from java source code. In *ASE ’06*, pages 123–134, 2006. IEEE Computer Society.

[15] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis. Design pattern detection using similarity scoring. In *Software Engineering*, pages 896–909. IEEE Transactions on, Nov. 2006.

[16] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR ’05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR’05)*, pages 134–142, 2005. IEEE Computer Society.