# BEHAVIOR-DRIVEN DESIGN PATTERN RECOVERY

Kamran Sartipi and Lei Hu
Department or Computing and Software, McMaster University
Hamilton, ON, Canada
email: {*sartipi, hul4*}*@mcmaster.ca*

**ABSTRACT**
In this paper, we present an approach for enhancing program understanding and reusability through a behavior-driven design pattern recovery process. In this context, incorporating behavioral features would characterize the approach as a goal-driven and scalable pattern recovery process. The proposed technique consists of a feature-oriented dynamic analysis and a two-phase design pattern detection process. The dynamic analysis operates on the system's scenario-driven execution traces and produces a mapping between features and their implementation at class level. For the two-phase design pattern detection, we employ approximate matching and structural matching algorithms to identify the instances of the target design pattern that is described using our proposed Pattern Description Language (PDL). The correspondence between system features and identified design pattern instances can facilitate the construction of more reusable and configurable software components to be maintained. We have implemented an Eclipse plug-in toolkit and have conducted experimentation on three versions of JHotDraw systems to evaluate our approach.

**KEY WORDS**
Dynamic Analysis; Behavior Feature; Design Pattern Detection; Feature-specific Scenario; Pattern Matching.

## 1 Introduction

Industrial software products in specific application domains are usually developed according to a reference architecture which consists of core parts and variable parts that are meant to satisfy the evolving requirements of the new products. Design patterns represent common solutions to design problems that allow reusability of design and contribute in reducing the system's complexity [11]. Moreover, the knowledge about design patterns within a software system can help the comprehension of the applied design decisions and adopted solutions made by the software designer. As a result, the recovery of design patterns is a crucial research problem that has gained lots of attention within software engineering community.

In this paper, we propose an approach based on a hybrid dynamic and static analysis to address the problem of reusing existing system's design that correspond to specific software behavior as the goal of the recovery process. Consequently, these patterns can be used in developing a family of similar systems that share the same core features, through: i) a dynamic analysis that allows us to restrict the analysis to the source code that implement specif features; and ii) a static analysis that identifies the instances of a design pattern that are specified using a proprietary pattern description language. A repository of different design pattern specifications allows us to identify the instances of different design patterns within the selected feature.

In this approach, the recovered patterns are not restricted only to the design patterns introduced by Gamma et al. [11], but they can be defined by the users. During the 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.

Figure 1 illustrates the proposed framework for behavior-driven design pattern detection. The framework consists of two parts: *feature-oriented dynamic analysis* and *two-phase design pattern detection process* which are described in the following sections.

## 2 Feature-oriented dynamic analysis

In this section, we locate the implementation of specific software features within the source code by the means of scenario execution, sequential pattern mining, and concept lattice analysis. In summary, the steps include: feature-specific scenario set selection and execution on the instrumented software system; extracting execution patterns from the execution traces; applying concept lattice analysis

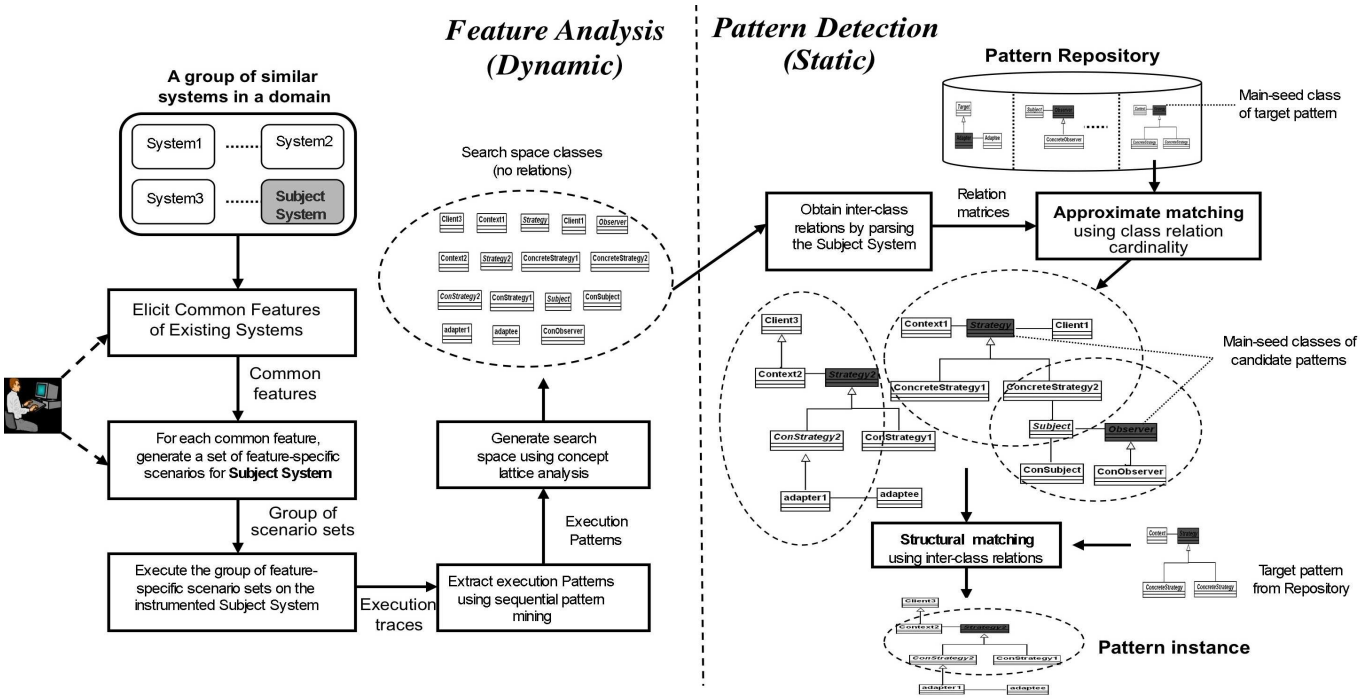



Figure 1. The proposed framework for behavior-driven design pattern detection.

on the execution patterns to assign classes to features and generate search space. In the remaining of this section the above steps are described in more detail.

### 2.1 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 each of which contains a specific software feature. We call this set of scenarios as *feature-specific scenario set*. As an example, for the feature "move" of the drawing tool JHotdraw5.1, we generate the following scenarios that all share the operation "move" to move a figure; these 5 scenarios constitute a feature-specific scenario set. Note that operations "start" and "exit" are not specific features, as they are repeated in almost all scenarios.

1 $start, drawarectangle, \textbf{\textit{move}}, exit$
2 $start, drawaellipse, \textbf{\textit{move}}, exit$
3 $start, drawapolygon, \textbf{\textit{move}}, exit$
4 $start, drawaline, \textbf{\textit{move}}, exit$
5 $start, insertaimage, \textbf{\textit{move}}, exit$

**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. TPTP is an open platform which provides the services such as application monitoring, tracing and profiling. 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 object invocations. We can enable the Filter Model option of the TPTP in order to eliminate the library class traces from the large traces of a scenario execution. Through setting the Filter in TPTP we can choose class names that we are interested to be profiled. We can also discard system libraries, e.g., Java system classes. Moreover, the preprocessing operation is applied on the execution traces to eliminate all the redundant object invocations caused by the cycles of the program loops. The trimmed execution traces are then fed into the next step, execution pattern generation.

**Execution pattern generation**
Finally, 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]. In our implementation, an *execution pattern* is defined as a contiguous sub-trace of an execution trace that exists in a number of execution traces. This strategy generates the meaningful execution patterns, each of which consists of a set of core classes that implement the specific feature of the subject system.

### 2.2 Execution pattern analysis

After we obtain the execution patterns of the specified feature, we use concept lattice analysis to cluster the group of classes in the execution patterns that exclusively correspond to the specified feature of a scenario set. A similar technique allows us to cluster the group of classes in pat-

terns that are common to every scenario set. In our setting for the concept lattice analysis, an *object* is a targeted feature $\phi_i$ which is shared within the feature-specific scenario set $S_\phi$, and an *attribute* is a class $c$ that participates in the execution patterns within $S_\phi$. In this context, the cluster of common classes appear at the upper region of the lattice, and clusters of feature-specific classes are located at the nodes in the lower region of the lattice. Thus, a mapping between a software feature and its implementation is obtained. The obtained classes are considered as the search space for the two-phase design pattern detection process which is discussed in the following section.

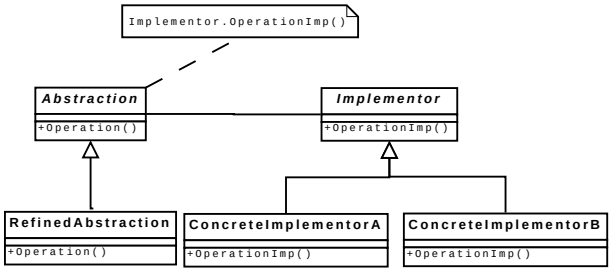
## 3 Two-phase design pattern detection

Design patterns represent high level concepts in object oriented methodology. Therefore, by identifying their occurrences in the implementation of a software feature, we are able to reveal some design aspects of that feature. Nevertheless, pattern detection is not a trivial task since we need to find all the possible pattern instances whose structures are consistent with the target pattern. Therefore, pattern detection is a complex and time-consuming operation that can easily result in combinatorial explosion [13]. To avoid this inherent complexity in pattern detection process, we first decompose the search space (i.e., the set of all classes in the subject system) into a group of smaller clusters of classes around each class $c$ of the search space, where class $c$ would become the *main-seed class* of a candidate instance pattern (i.e., the cluster of classes around main-seed class). Second, we perform two pattern matching algorithms (i.e., approximate matching and structural matching) to obtain all the pattern instances in the search space that match with the target pattern.

### 3.1 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**
A critical parameter in pattern detection process is the size of the search space for large software systems. A brute force approach to identify all the pattern instances in a system with a large number of classes will result in a combination explosion due to multiple roles that classes can play in the pattern. 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




```
1   Begin-PDL
2     Pattern : "Bridge"
3       Main-seed class : "Implementor"
4         Depth_1 :
5           Inherited_by :
6             "ConcreteImplementorA";
7             "ConcreteImplementorB"
8           in_Association :
9             "Abstraction"
10          in_Delegation :
11            "Abstraction"
12        Depth2 :
13          Seed-Depth_2 : "Abstraction"
14            Inherited_by :
15              "RefinedAbstraction"
16    End-Pattern
17  End-PDL
```

Figure 2. Class diagram and PDL description of Bridge design pattern.

generate a short list of approximately similar instance patterns to the target pattern. The *main-seed attributes* are as follows:

- number of *Inherit_From* relations
- number of *Inherited_By* relations
- number of *Association* relations
- number of *Abstract* relation (0 or 1)

Figure 2 illustrates the class diagram and PDL specification of the *Bridge* design pattern which consists of a main-seed class and two *Depths* classes around the main-seed class. The rational for such two-depth design pattern specification is as follows. We observed that for all the GoF design patterns presented in [11], there exist one or more classes which can reach any other class in the pattern within a shortest path of 2 edges. We name this set of classes as potential main-seed classes. During the structural matching phase, one of these classes is selected as the main-seed according to the largest number of class-relationship that a candidate main-seed possess within the pattern (namely degree of class). Using the same approach the candidate main-seeds within a search space (i.e., class cluster) are determined. In Figure 2 lines 4 to 11 describe the structural relations between main-seed class and other classes in

depth1, while lines 12 to 15 indicate the structural information between *Seed-Depth_1* classes and depth2 classes in the pattern.

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 1 "ApproximateMatching" below receives the search space, class relations, 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 a class $c_i$ from the search space.

**Algorithm 1**: ApproximateMatching

**Input**:
$SP$: search space (set of classes obtained from feature analysis)
$M$: inter-class relations matrices
$t$: target pattern from pattern repository
$tsh$: cut-off threshold similarity value
**Local Variable:**
$Attr(c_i)$: vector attribute of $c_i$
$Attr(c_t)$: vector attribute of $c_t$
$p_i$: a candidate instance pattern
**Result**:
$LE$: list of eligible instance patterns
**begin**
  $LE := \emptyset$ ;
  $c_t := GetMainSeedClass(t)$;
  $Attr(c_t) := ComputeAttrValue(t)$ ;
  **for** *class* $c_i \in SP$ **do**
    $Attr(c_i) := ComputeAttrValue(c_i, M)$ ;
    **if** $sim_{apx}(Attr(c_t), Attr(c_i)) \geq$ *tsh* **then**
      $p_i := GeneratePattern(c_i)$;
      $LE := Add(LE, p_i)$;
**end**

**Structural matching**
Structural matching algorithm (presented as Algorithm 2) 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 relations. 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. Function $Depth1Matching()$ and $Depth2Matching()$ are discussed below in more details:

**Algorithm 2**: StructuralMatching

**Input**:
$p_i$: a candidate instance pattern
$t$: target pattern from pattern repository
$M$: inter-class relations matrices
**Local Variable:**
$D1_t$: set of classes in depth1 of target pattern
$D1_{mch}$: set of combinations of depth1 matched classes
$D2_t$: set of classes in depth2 of target pattern
$C_j$: a combination of depth1 classes
**Result**:
$R$: set of identified instance patterns
**begin**
  $R := \emptyset$ ;
  $D1_t := GetDepth1Classes(t)$;
  $D2_t := GetDepth2Classes(t)$;
  $D1_{mch} := Depth1Matching(t, D1_t, p_i, M)$;
  **for** $C_j \in D1_{mch}$ **do**
    $R :=$
    $R \cup Depth2Matching(C_j, D2_t, p_i, M)$;
**end**

• $Depth1Matching()$: using candidate instance pattern $p_i$ and relation matrices $M$, all the depth1 classes of $p_i$ are obtained. Through listing all the combinations of these depth1 classes, we obtain the instances of the target pattern in depth1. Here we exclude those instances where one class holds two roles in one pattern.

• $Depth2Matching()$: for each combination of depth1 classes (obtained above) we identify the depth2 classes of the candidate instance pattern according to the inter-class relation $M$. The comparison of the combination of depth1 and depth2 of the $p_i$ with those of target pattern $t$ results in the matched instance pattern for target pattern.

After a pattern instance is detected, a further manual verification has to be performed to check whether the pattern is really implemented within the subject system through browsing the source code and comparing with the result of other similar pattern detection tools [18].

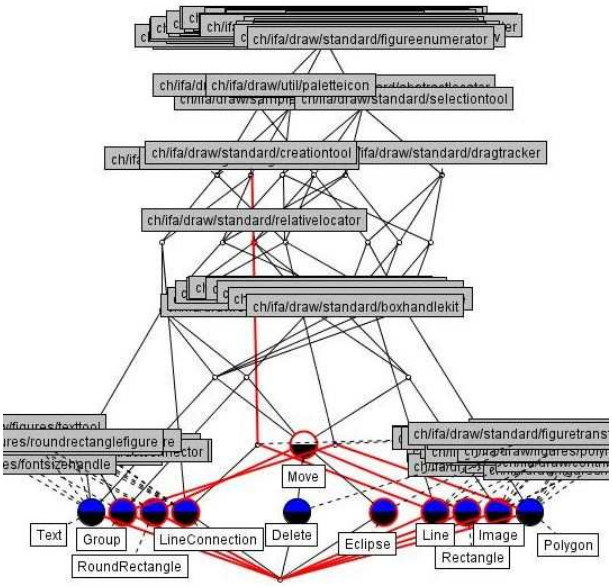



Figure 3. Concept lattice representation of the features and classes. Each bubble represents a feature and the shaded labels represent classes. Specific classes and common classes are clustered at the lower region and upper region, respectively.

## 4 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. Moreover, the designers of JHotDraw provide the usage descriptions of the design patterns in the software documentation, which allow us to evaluate the results of our approach. Based on discussions in the earlier sections, we apply our proposed approach on three versions of JHotDraw, ver5.1, ver6.0b1 and ver7.0.7 to extract reusable software artifacts.

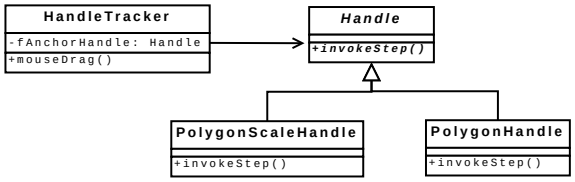



Figure 4. A Strategy pattern instance of feature Drawing Polygon in JHotDraw 5.1.

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. The case study has been performed in accordance with the proposed framework and includes the following major steps: selecting features, generating feature-specific scenario sets, sequential pattern mining, concept lattice analysis, and pattern matching.

[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.

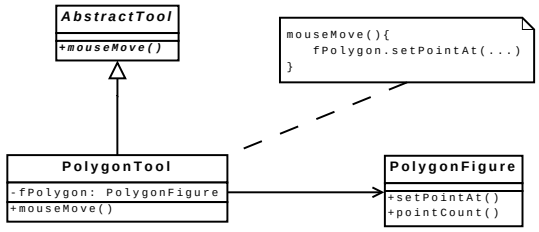



Figure 5. Adapter design pattern instances of feature Drawing Polygon in JHotDraw 5.1, 6.0b1 (for 7.0.7 is slightly different, it is not shown for space consideration).

Table 1 depicts the experimental results of execution trace extraction and execution pattern mining for 10 features of the three versions of JHotDraw systems. In a further step, we supply the resulting execution patterns obtained from the sequential pattern mining to a concept lattice generation tool, ConExp [3]. The resulting concept lattice is shown in Figure 3 where the feature-specific concepts are gathered at the lower part of the lattice. 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.

In the following phase of the experimental study, we apply pattern detection algorithms "ApproximateMatching()" and "StructuralMatching()" (discussed in Section 3) on the search space to detect all the pattern instances of the target patterns in the pattern repository. Currently, our pattern repository contains the following patterns: *Adapter, Proxy, Observer, Decorator, Bridge* and *Strategy & State*. We describe structural information of each pattern using the proposed pattern definition language (PDL) and store it into the pattern repository. Figure 2 illustrates the structural information of Bridge design pattern in PDL. 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. Figure 4 shows an instance of Strategy design pattern that is detected from JHotDraw 5.1. This pattern instance is related with the feature *Drawing Polygon* since class *PolygonHandle* and *PolygonScaleHandle* are highly related classes of this feature.

The results obtained from the two-phase pattern detection process can support the task of migrating an existing family of software systems into a software product line. For example, Figure 5 presents the detected Adapter design pattern instances of feature *Drawing Polygon* in JHotDraw 5.1 and 6.0b1 (for space consideration the result for 7.0.7

| Specific Feature of JHotDraw | Number of Scenarios | Average Trace Size | Average Pruned Trace Size | Number of Extracted Patterns | Average Pattern Size |
|---|---|---|---|---|---|
| Rectangle | 4 / 4 / 4 | 2494 / 4889 / 11962 | 927 / 2165 / 2110 | 13 / 24 / 23 | 126 / 170 / 220 |
| Round Rectangle | 4 / 4 / 4 | 2369 / 5040 / 10620 | 927 / 2327 / 1864 | 15 / 25 / 19 | 153 / 138 / 183 |
| Ellipse | 4 / 4 / 4 | 2104 / 5492 / 10580 | 773 / 2226 / 1915 | 15 / 22 / 24 | 112 / 185 / 175 |
| Polygon | 4 / 4 / 4 | 4553 / 15769 / 17130 | 1654 / 4029 / 3142 | 21 / 41 / 38 | 199 / 192 / 130 |
| Line | 4 / 4 / 4 | 1439 / 4253 / 9882 | 546 / 2224 / 2123 | 7 / 24 / 27 | 157 / 170 / 126 |
| Move | 4 / 4 / 4 | 2599 / 4930 / 11341 | 774 / 2688 / 2487 | 18 / 34 / 52 | 31 / 89 / 37 |
| Delete | 4 / 4 / 4 | 1323 / 5739 / 8540 | 623 / 2456 / 969 | 16 / 32 / 24 | 36 / 89 / 49 |
| Group | 5 / 5 / 5 | 4579 / 12978 / 33921 | 1397 / 4675 / 4842 | 36 / 66 / 57 | 26 / 85 / 49 |
| LineConnection | 4 / 4 / 4 | 5238 / 10356 / 24075 | 1681 / 4158 / 4437 | 38 / 53 / 56 | 36 / 78 / 73 |
| Text | 4 / 4 / 4 | 1524 / 6074 / 18629 | 781 / 2435 / 2204 | 11 / 35 / 12 | 62 / 105 / 288 |

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

Table 1. Results of execution trace extraction and execution pattern mining for 10 features of three versions of JHotDraw systems.

| Design Pattern Detection Techniques | Category of Techniques | Coarse Matching | Fine Matching | Pattern Description Language | Supporting User -Defined Pattern | Feature -Oriented |
|---|---|---|---|---|---|---|
| DP-Miner [8] | matix-based | NA | NA | NA | NA | NA |
| PINOT [17] | structure-based | NA | Yes | NA | NA | NA |
| Balanyi [7] | structure-based | Yes | Yes | DPML | Yes | NA |
| Nikolaos [18] | matix-based | NA | Yes | NA | NA | NA |
| Lucia [15] | structure-based | Yes | Yes | VL | Yes | NA |
| Antonial [6] | metric-based | Yes | NA | AOL | NA | NA |
| Yann-Gael [13] | metric-based | Yes | Yes | NA | NA | NA |
| DPVK [19] | structure-based | Yes | NA | RSF & REQL | NA | NA |
| PAT [14] | structure-based | Yes | NA | PROLOG | NA | NA |
| **Our technique** | **structure-based** | **Yes** | **Yes** | **PDL** | **Yes** | **Yes** |

Table 2. Comparison of several design pattern detection techniques

is not shown). By comparing and analyzing the detected pattern instances of the three versions JHotDraw systems, we notice that the implementation of the feature *Drawing Polygon* in JHotDraw 5.1 and 6.0b1 is very similar, while there exist some differences between the implementation in JHotDraw 7.0.7 with that in JHotDraw 5.1 and 6.0b1. Performing such comparison and analysis on the detected pattern instances of these 10 features, which are used most frequently in the applications and shared by all the three versions of JHotDraw systems, can help to comprehend the features' implementation at the design level and allows for a quick understanding of evolution of the features within the software.

## 5 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. [20] 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. [17] 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. [15] 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. [18] 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.

Several other approaches on design pattern detection have also experimented with JHotDraw system [5, 8, 13, 15, 17, 18]. We compared the results obtained by these approaches including our approach which indicates some differences in aspects such as: description, completeness, and variations of design patterns. In Table 2 we compare our technique with several current major design pattern detection techniques based on different criteria such as: type of detection algorithm (coarse or approximate matching versus fine or detailed matching); kind of pattern description language; supporting user defined patterns; and using software features in pattern detection process.

## 6 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 applied our technique on three versions of a software system as members of a software family. In this context, the common design patterns that implement specific software features are candidates to construct a common reference architecture for the product line. The major parts of the proposed approach are summarized as follows. The first phase (behavior feature analysis) consists of feature and scenario identification, execution pattern mining, and concept lattice analysis to extract both feature specific and common groups of classes that implement features of the system. This phase concludes by producing a search space for the matching process. In the second phase (pattern detection), first a target design pattern is defined using a novel pattern definition language that uses a center-role main-seed class and two-level surrounding classes. The matching process consists of an approximate matching and a structural matching that precisely identify the specified pattern while providing scalability of the process. We have successfully experimented with JHotDraw Java GUI system which is considered as a benchmark for design pattern recovery. Our results conform with those approaches that provided their results publicly. Finally, the proposed approach has been implemented as a plug-in in the Eclipse open platform.

## 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 *IEEE ICDE '95*, pages 3–14, Washington, DC, USA, 1995.

[5] H. Albin-Amiot et al. Instantiating and detecting design patterns: Putting bits and pieces together, 2001.

[6] G. Antoniol et al. Design pattern recovery in object-oriented software. In *IEEE IWPC '98*, pages 153–160, 1998.

[7] Z. Balanyi and R. Ferenc. Mining design patterns from c++ source code. In *IEEE ICSM '03*, page 305, Washington, DC, USA, 2003.

[8] J. Dong, D. S. Lad, and Y. Zhao. Dp-miner: Design pattern discovery using matrix. In *Proceedings of IEEE ECBS' 07)*, pages 371–380, Washington, DC, USA, 2007.

[9] T. Eisenbarth, R. Koschke, and D. Simon. Derivation of feature component maps by means of concept analysis. In *IEEE CSMR'01*, 2001.

[10] M. El-Ramly, E. Stroulia, and P. Sorenson. Recovering software requirements from system-user interaction traces. In *SEKE '02*, pages 447–454, New York, NY, USA, 2002.

[11] E. Gamma et al. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[12] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verla, 1999.

[13] Y.-G. Gueheneuc, H. Sahraoui, and F. Zaidi. Fingerprinting design patterns. In *IEEE WCRE'04*, pages 172–181, Washington, DC, USA, 2004.

[14] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *IEEE WCRE'96*, pages 208–215, 1996.

[15] A. D. Lucia et al. A two phase approach to design pattern recovery. In *IEEE CSMR'07*, pages 297–306, Amsterdam, Netherlands, 2007.

[16] K. Sartipi and H. Safyallah. Application of execution pattern mining and concept lattice analysis on software structure evaluation. In *SEKE*, pages 302–308, 2006.

[17] N. Shi and R. A. Olsson. Reverse engineering of design patterns from java source code. In *IEEE ASE '06*, pages 123–134, Washington, DC, USA, 2006.

[18] N. Tsantalis et al. Design pattern detection using similarity scoring. In *IEEE TSE*, pages 896–909, 2006.

[19] W. Wang and V. Tzerpos. Design pattern detection in eiffel systems. In *IEEE WCRE'05*, pages 165–174, 2005.

[20] A. Zaidman et al. Applying webmining techniques to execution traces to support the program comprehension process. In *IEEE CSMR'05*, pages 134–142, 2005.