An Eclipse plug-in for the Detection of Design Pattern Instances through Static and Dynamic Analysis

Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, Michele Risi
Dipartimento di Matematica e Informatica
Università degli Studi di Salerno
Fisciano(SA), Italy
{adelucia, deufemia, gravino, mrisi}@unisa.it

Abstract—The extraction of design pattern information from software systems can provide conspicuous insight to software engineers on the software structure and its internal characteristics. In this demonstration we present ePAD, an Eclipse plug-in for recovering design pattern instances from object-oriented source code. The tool is able to recover design pattern instances through a structural analysis performed on a data model extracted from source code, and a behavioral analysis performed through the instrumentation and the monitoring of the software system. ePAD is fully configurable since it allows software engineers to customize the design pattern recovery rules and the layout used for the visualization of the recovered instances.

Keywords— reverse engineering; design pattern recovery; Eclipse plug-in; source code analysis.

I. INTRODUCTION

Recovering design patterns from source code represents an important step in software maintenance because it provides information relevant to understand the rationale behind the system design, to improve the system documentation, and to guide the restructuring of the system [1][9][22]. Therefore, automating the process of design pattern detection is highly desirable to facilitate program comprehension activities.

In the recent years, several design pattern recovery approaches and tools supporting them have been provided in the literature (e.g., [1][9][10][15][22]). In [4] we have evaluated and classified them taking into account the employed pattern identification strategy, the representation used for coding design patterns, the kind of support they provide for recognition (i.e., manual, semi-automatic or automatic pattern recovery), the type of design patterns they are able to recover, the software analyzed to assess the effectiveness of the proposed pattern recovery strategies.

Recent studies have also demonstrated that layout plays an important role in the comprehension of software maintenance tasks (e.g., [19]). Moreover, the layout of design patterns is particularly important since appropriate representation can help engineers to accurately identify the correct solution in less time and effort [12][18]. In particular, the results reported in [18] revealed that the use of layout scheme closed to the classical design pattern representation [7] significantly reduces the time taken by students to identify roles in design patterns.

In this demonstration we present ePAD (eclipse plug-in for design Pattern Analysis and Detection), a tool that integrates recovery and layout functionalities for design patterns. In particular, ePAD detects design pattern instances from OO source code through a static analysis, to extract the instances according to their structural properties [4], and a subsequent dynamic analysis, to verify the runtime behavior of the detected instances [5]. ePAD has been implemented as an Eclipse plug-in and provides several visualization functionalities for the analysis of the results of the recovery process. In particular, it provides multi-layouts to visually represent the recovered instances and to access their properties. ePAD is fully customizable since it allows engineers to configure the definition of the patterns structure and their behavior and the layout to be used for visualizing their instances. In order to highlight the main features of ePAD we present an example of usage of the tool on JHotDraw 5.1 and discuss the obtained results.

This paper is organized as follows. Section II presents ePAD and the recovery process underlying it. Sections III and IV describe a usage scenario of ePAD and its configuration. The results obtained from a case study on JHotDraw are summarized in Section V. Section VI concludes the paper.

II. THE TOOL

In this section we present the recovery process of ePAD and an overview of the Eclipse plug-in implementing it. ePAD is downloadable from http://www.sesa.dmi.unisa.it/ePAD/

A. ePAD Recovery Process

ePAD identifies instances of structural and behavioral design patterns from Java code by applying static and dynamic analyses. In particular, Fig. 1 shows the activity diagram describing the recognition process of ePAD, where rectangles represent data and rounded rectangles represent phases.

During the Preliminary Analysis information proper to
recover design pattern instances is extracted from input source code and stored in a repository by using the Source Code Extractor. In particular, class diagram information, e.g., the name and type of classes, inheritance and association relationships are exploited to construct the corresponding UML class diagram and stored to be used for the Model Analysis. Information on method declarations and invocations useful to perform the Source Code Analysis are also stored.

During the Structural Analysis the instances of design patterns are identified by analyzing the class diagram structure. This recovery process is organized in two steps. In the first step (Model Analysis), the candidate design patterns are identified at a coarse-grained level by analyzing the class diagram information obtained during the Preliminary Analysis and exploiting the Design Pattern Library. In particular, the problem of design pattern recovery is reduced to the problem of recognizing subsentences in the UML class diagram, where each subsentence corresponds to a design pattern instance [4]. In the second step (Source Code Analysis), a fine-grained source code analyzer checks if the identified candidate patterns are correct instances or false positives. This is accomplished by verifying at source code level the declarations and the invocations of the methods of the classes involved in the candidate instances. The definition of the recognition algorithms for these phases can be found in [4].

The set of candidate instances obtained from the Structural Analysis are then given as input to the Behavioral Analysis, together with the specification of the pattern behaviors (i.e., Design Pattern Library), the executable program with a test suite (i.e., Bytecode and Test Cases). The goal is to identify the candidates having a behavior that complies with the pattern definition they are instance of. This is performed in three steps. First, the bytecode of the classes involved in the candidate instances is instrumented (Instrumentation step). The files obtained from instrumentation are executed together with the program to monitor the method calls of the candidate instances on a suitable test suite (Monitoring step). Finally, the obtained method trace is validated by a parser able to recognize the sequence of method calls defining the design pattern behavior (Validation step). Details of the three steps can be found in [5].

B. ePAD Implementation

We decided to implement ePAD as an Eclipse plug-in in order to enhance ePAD’s extensibility and take advantage of the latest development techniques [8]. The plug-in exploits the code analysis tool Source Navigator [20] in order to implement the Source Code Extractor module in Fig. 1. The information on the source code is structured as a set of tables storing information on classes, relationships between classes, method declarations, method invocations, and so on.

The Structural Analysis of the plug-in has been implemented as a parser that takes as input the set of tables obtained from the Source Code Extractor module and produces as output a set of textual descriptions representing the candidate design pattern instances [4]. In particular, the output descriptions include the qualified name of the classes involved in the pattern instances together with the low-level checks to be performed on them. A source code analyzer invokes the methods accomplishing fine-grained checks on the classes involved in the pattern instances. The pattern descriptions that passed these checks are the output of the Structural Analysis phase and the input of the Behavioral Analysis phase.

![Figure 1. The recovery process of ePAD](image)

To instrument the bytecode of the classes involved in the candidate instances we use the Eclipse plug-in Probekit [16], developed within the TPTP (Test and Performance Tools Platform Project) [21]. Probekit allows users to write Java code fragments, named probes, that can be injected at specified points in the classes, such as method entry, method exit, class loading, in order to collect runtime data about objects, instance variables, arguments, exceptions, and so on. To trace the invocation of methods at run-time for each candidate ePAD generates a probe file, which is able to inject additional Java code at method entry and method exit of all classes of the candidate. The Monitoring is accomplished by executing the instrumented Java program on the set of specified test cases. The output is the sequence of all monitored method calls and the order in which they are run, which is then validated against the pattern behavioral specification.

III. EXAMPLE OF USAGE

In this section we describe an example of the usage of ePAD for retrieving and analyzing design pattern instances from JHotDraw 5.1 [13]. JHotDraw has been chosen since it was originally developed to illustrate the good use of design patterns for designing and documenting systems [14]. Thus, it is an ideal candidate to verify the effectiveness of a design pattern recovery approach as demonstrated by its use in the evaluation of many recovery tools.

Table I illustrates each step a user has to carry out in order to accomplish the tasks of design pattern recovery and analysis of the results. In the following we detail each step.
### TABLE I. USAGE STEPS OF EPAD

<table>
<thead>
<tr>
<th>STEPS</th>
<th>EXPECTED OUTPUT</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection of the software system to be analyzed</td>
<td>The diagram view visualizes the UML class diagram representing the structural information of the selected software system</td>
<td>This step provides to the user a graphical representation of the software systems, which will be exploited to visualize the results of the recovery process</td>
</tr>
<tr>
<td>Execution of the Structural Analysis phase of the recovery process</td>
<td>A list of the design pattern instances recovered according to the specified structural properties</td>
<td>The recovered instances are textually listed in a table by providing for each class involved in a pattern the role it plays. The user can also visualize an instance in the class diagram shown in diagram view</td>
</tr>
<tr>
<td>Execution of the Dynamic Analysis phase of the recovery process on the candidate instances of behavioral patterns</td>
<td>The list of the design pattern instances that have shown a behavior complying with the pattern definition they are instance of during the execution of the test cases</td>
<td>Probekit is used to monitor the execution of the instrumented candidate instances on a set of test cases</td>
</tr>
<tr>
<td>Analysis of the recovery results and pruning of the false positive instances</td>
<td>The recovered design pattern instances are visualized in the diagram view</td>
<td>Through the visualized of the pattern instances it is possible to access and refactor the code of the classes involved in the instances. The user can also prune the false positives from the list of recovered instances</td>
</tr>
</tbody>
</table>

#### A. UML Class Diagram Construction

Before performing structural analysis the source code is analyzed to extract information needed to construct the UML class diagram. To this end, the user has to first select the folder containing the source code of the software system (Select folder button in Fig. 2). Then, ePAD automatically creates an Eclipse project containing a copy of the selected source code and the class diagram is built by clicking the Class Diagram button in Fig. 2. ePAD makes use of the GMF framework to visualize the constructed class diagram in the diagram Eclipse view (see Fig. 2).

![Image](image-url)

**Figure 2.** Visualization of the selected software system as UML class diagram
B. Structural Analysis

The Structural Analysis is performed by executing a parser on the constructed class diagram [4]. In particular, the analysis is carried out in two steps: applying the coarse-grained rules involving class diagram model (Model Analysis button in Fig. 2) and checking fine-grained rules on the source code (Source Code Analysis button in Fig. 2). The UML Class Diagram Construction and Structural Analysis phases can also be executed in one step by clicking the fourth button on the toolbar of the ePAD view at the bottom of Fig. 2.

Fig. 3 shows the results of the structural analysis for JHotDraw 5.1. The statistics on the recovered instances are textually visualized in the Pattern Instances tab, providing for each class the role it plays. By selecting an instance, the involved classes are highlighted in the diagram view as UML class diagram. As an example, Fig. 3 shows the layout of an instance in the list of retrieved Adapter pattern instances. The colors used to visualize the diagram can be configured by using the color palette associated to the ePAD view.

C. Behavioral Analysis

Since the behavioral analysis requires the execution of the software under analysis, the user has first to select the folder containing the bytecode files (Select File button in Fig. 2). Then, when the Profiling button is clicked the Probekit plug-in is activated, which requires the selection of the test cases to be executed on the software system. To accomplish monitoring, ePAD automatically generates a probe file for each candidate and invokes Probekit to execute the instrumented Java program on the set of specified test cases. For each probe file, the monitoring process produces an XML file containing the sequence of monitored method calls and the order in which they are run. Such a file describes the behavior of a candidate pattern instance on a test case. Thus, to verify whether the behavior of an instance complies to the definition of pattern’s behavior a pattern matching algorithm is executed (Sequence Analysis button in Fig. 2). In particular, such algorithm looks for a sequence of messages (in the XML file) exchanged between the objects involved in an instance that satisfies the pattern behavior. The resulting instances are textually visualized in the results view as done for the structural instances. Fig. 4 shows the profiling environment with the frame for selecting the candidates to be monitored.

D. Visualization and Navigation of the Results

ePAD provides several visualization features for supporting the users in the analysis of the recovery results. In particular, the recovered design pattern instances can be visualized as class diagrams in the diagram view by using different styles: visualization of a single instance; highlighting a single instance in the whole class diagram with zoom-in zoom-out features; visualization of a design pattern instance with its relationships with other instances. As an example Fig. 3 shows the visualization of a single instance of Adapter pattern. Notice that the classes involved in the pattern are colored with the same color (e.g., AbstractHandle is colored in white and does not play a role in the design pattern) and grouped in boxes according to the role they play. Once a pattern instance is visualized in the view, the users can access the code of the classes involved in the instances to analyze and/or refactor it, by clicking the first button on toolbar of the ePAD view.

When the user analyzes the instances obtained from the recovery process, s/he can classify an instance as false positive by marking the check box Not Valid in the Pattern Instances tab, as shown in Fig. 3. The false positive instances are then highlighted in red and not considered in the summary statistics of the results.
IV. DESIGN PATTERN CONFIGURATION

ePAD allows users to choose the design patterns to be retrieved from the source code through the ePAD Preferences page. In this page the user can also edit the design pattern recovery rules, having the possibility of considering variants of patterns in the recovery process. In particular, the user can define the structural and/or behavioral rules of a pattern, as well as, the layout used to show the recovered instances as UML class diagram.

The structural properties are specified by coarse grained rules defining the relationships between classes or interfaces and fine grained rules exploiting the algorithms EXTEND, INHERITANCE, and DELEGATION described in [3]. As an example, the rule for the State pattern shown in Fig. 5 specifies that a CALLS relationship holds between Context represented by \( x \) and State represented by \( y \) and an EXTENDS relationship holds between State or ConcreteState represented by \( y \) and \( z \), respectively. The behavioral rules are defined as a set of sequences, each describing the message exchanged by objects. As an example, sequence1 in Fig. 5 specifies that a client invokes a method denoted by label \( A \) of Context that subsequently invokes a method (i.e., \( B \)) of ConcreteState or State and then one of its methods (i.e., \( D \), different from method with label \( B \)).

Layout information are specified in XML to define how the elements involved in pattern instances have to be arranged as UML class diagram. As an example, Fig. 6 shows the specification for the classical Adapter pattern representation.

The definition of the languages used to specify the structural and behavioral rules, and layout information are out of the scope of this demonstration.

V. CASE STUDY RESULTS

In this section we report on the results obtained by ePAD on JHotDraw 5.1. This code included 8300 LOC, 155 Classes, 4612 number of delegations and inheritances, and 98 number of classes involved in the recovered design pattern instances. Table II shows the results we obtained with Structural Analysis, reporting on the number of instances recovered with Model Analysis and Source Code Analysis, and precision statistics. The results achieved with Behavioral Analysis are shown in Table III, reporting on the number of structural pattern candidates received as input, the number of test cases executed, the number of recovered instances, and precision statistics. Note that we do not report on recall since the documentation does not provide a clear indication of the use of some design patterns. The test cases executed during behavioral analysis are those included in the JHotDraw release.
Observe that we performed several case studies to assess the effectiveness of the proposed design pattern recovery approach, which are reported in [4][5]. We also compared our results with those obtained by other approaches on the same software systems (those available on the Web or in articles). The analysis revealed that in general our tool was able to obtain good results. Furthermore, it outperformed other approaches in terms of precision on some patterns, e.g., Adapter and Observer patterns [4][5].

### TABLE II. RESULTS OF THE STRUCTURAL ANALYSIS

<table>
<thead>
<tr>
<th></th>
<th># instances from Model Analysis</th>
<th># instances from Source Code Analysis</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>408</td>
<td>36</td>
<td>92%</td>
</tr>
<tr>
<td>Bridge</td>
<td>404</td>
<td>44</td>
<td>73%</td>
</tr>
<tr>
<td>Composite</td>
<td>233</td>
<td>6</td>
<td>33%</td>
</tr>
<tr>
<td>Proxy</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>Decorator</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Observer</td>
<td>3798</td>
<td>3741</td>
<td>0.1%</td>
</tr>
<tr>
<td>Strategy</td>
<td>1337</td>
<td>614</td>
<td>3%</td>
</tr>
<tr>
<td>State</td>
<td>1337</td>
<td>614</td>
<td>0.3%</td>
</tr>
</tbody>
</table>

### TABLE III. RESULTS OF THE BEHAVIORAL ANALYSIS

<table>
<thead>
<tr>
<th></th>
<th># Candidates</th>
<th># Test Cases</th>
<th># Instances</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observer</td>
<td>3741</td>
<td>4</td>
<td>9</td>
<td>55%</td>
</tr>
<tr>
<td>Strategy</td>
<td>614</td>
<td>4</td>
<td>43</td>
<td>37%</td>
</tr>
<tr>
<td>State</td>
<td>614</td>
<td>4</td>
<td>32</td>
<td>4%</td>
</tr>
</tbody>
</table>

### VI. CONCLUSIONS AND FURTHER WORK

In this demonstration we have presented ePAD, an Eclipse plug-in for the detection and analysis of design pattern instances. It provides functionalities for the configuration of design pattern’s definition and visualization. To the best of our knowledge only RECLIPSE [3] and Ptidej [17] provide configuration functionalities similar to the ones supported by ePAD. In particular, RECLIPSE allows to graphically specify the pattern’s structure and, where reasonable, its behavior, and to inspect the obtained candidate instances. Ptidej provides a set of tools to evaluate and enhance the quality of object-oriented programs. Among its functionalities, it supports the specification of design patterns and the identification of their instances through a constraint solver [9]. However, both Ptidej and RECLIPSE do not allow users to specify layout information to be exploited for the visualization of the design pattern instances. This is particular important since diagram layout can significantly affect the comprehension of the pattern instances [18]. Other design pattern recovery tools mainly includes features to detect and list recovered design pattern instances without visualization support. As an example, the design pattern detection tool presented in [22] provides users with a simple GUI allowing to select the software system to be analyzed and generate a list of the recovered design pattern instances, whereas tools like PINOT [15], SanD [10], and PRAsisster [11] works at command line.

As future work we intend to extend the tool to support the visual specification of the pattern detection rules together with the layout information. An early prototype supporting the visual specification of pattern recovery rules through visual grammar productions was developed in the past [2]. However, this tool suffered of scalability issues and the specification of the grammar rules were too complex for users. Finally, we plan to extend ePAD by integrating the model checking approach proposed in [6].

### REFERENCES


