Improving Behavioural Design Patterns Detection Through The Incorporation Of User Knowledge

Mohammad H. Alshira'h, Saleh Alqatan

Abstract: Design pattern detection is useful for a range of software comprehension and maintenance tasks. Current tools are limited to a combination of static and dynamic analysis, which can lead to inaccurate results for behavioural design patterns, which are intrinsically dynamic. This work proposes a technique to address these limitations by enabling the user to augment the automated analysis with their expert knowledge. This can be used to iteratively refine the results, removing any inaccurate patterns. The evaluation on JHotDraw indicates that the approach can yield significant improvement in accuracy from a relatively small amount of input from the user.

Index Terms: Software Maintenance, Reverse Engineering, Design Pattern Detection, Design Recovery, Reengineering, Software Comprehension, Static Analysis, Dynamic Analysis

1 INTRODUCTION

A substantial amount of research effort has been applied to develop and implement tools to understand software systems without appropriate documentation. Design pattern detection approaches have utilised the source code as a main source of information. The typical detection process has three key steps. Firstly, it starts by parsing the source code into an intermediate representation. Secondly, it conducts a static or dynamic analysis to extract relevant facts. Finally, the detection algorithm works by matching between the intermediate representation of the facts, and the formal specification for each design pattern [1]. Reverse engineering is the idea of reconstructing the design of existing software systems. It is focused on the task of understanding program code without having precise documentation [2]. A design pattern [3] is a description or template for how to solve recurring software problems that can arise in many different situations. Design patterns can be detected from the source code. The automatic detection of design patterns is difficult. Each design pattern incorporates structural and behavioural components [4]. Discovering structure and behaviour components for each design patterns require both static and dynamic analysis. This paper proposes a user-driven approach to pattern detection. The motivation is that a limited amount of user input can complement the automated analyses, to produce results that are more accurate. We propose a framework to facilitate this, enabling the user to supply domain knowledge in the form of constraints, provide a means for the user to integrate this knowledge, and consider how this knowledge has an impact on the detection process.

The rest of this paper is organised as follows: Section 2 discusses some main concepts related to design pattern detections. Section 3 presents the proposed technique. Section 4 covers the JHotDraw case study. Section 5 discusses the related work. Section 6 covers the conclusion and future work.

2 BACKGROUND

A substantial amount of research effort has been applied to develop and implement tools to understand software systems without appropriate documentation. Design pattern detection approaches have utilised the source code as a main source of information. The typical detection process has three key steps. Firstly, it starts by parsing the source code into an intermediate representation. Secondly, it conducts a static or dynamic analysis to extract relevant facts. Finally, the detection algorithm works by matching between the intermediate representation of the facts, and the formal specification for each design pattern. Design patterns [3] are a solution for frequent software problems, which make the maintenance of software easier and assist in the acquisition of additional new requirements. The Gang-of-Four (GoF) patterns [3] are the most widely used classification. This categorisation presents 23 design patterns, categorised according to their purpose into structural, creational and behavioural patterns. Each pattern is expressed using two components, structural and behavioural, where each pattern has a particular abstract model specification that expresses a pattern representation and has unique intent. The difficulty of extracting patterns is not the same for all pattern categories. For structural design patterns, most information can be recognised in inter-class relationships, which can be trivially extracted from the source code. On the other hand, behavioural patterns are considered the most difficult pattern category to be detected [5], since these rely on the availability of a representative set of program executions.

2.1 Pattern Detection as Constraint Satisfaction Problem

The design pattern detection process can be considered as a constraint satisfaction problem (CSP) [6]. A CSP [7] is represented as a set of variables and set of constraints that bounds the combination of values of the variables in a specified domain. It is denoted by (V, D, C) where V is set of variables taking their value in a specified domain D upon a set of defined constraint C. The solution for a CSP is a complete assignment of variables, where each variable assign to a value from the domain when the constraints hold. From the pattern detection perspective [6], the domain is the source code of the analysed system, and the constraints are design pattern specification.
models. The solution of such problem leads to find a class structure that matches the desired design pattern.

2.2 Motivating Scenario

Current design pattern detection tools are based entirely on static and dynamic code analysis. This becomes problematic when two different patterns share the same structure with different intents. When this is the case, current tools become inaccurate, reporting selections of source code as the manifestation of multiple possible patterns instead of just one. This is the case for patterns such as State, Strategy, Bridge, Chains of Responsibility, Decorator, and Proxy [3]. An example is shown in the Figure 1, where the Command and Adapter patterns share the same structure. Specifically, they have a super-class with a set of subclasses, each of which is associated with an external class. The Command pattern [3] provides an interface that manages operations, encapsulating all information as one object to be able to call other methods. The main participating roles include Command, Client, Invoker and Receiver. Invoker determines when the method will be called, and the Receiver performs the operations. In more detail, the abstract class Command, with a method execute(), has a number of Concrete classes ConcreteCommand, which override the execute() operation. The class Invoker aggregates with the Command class and keeps references for all derived ConcreteCommand classes. Each ConcreteCommand class encapsulates Receiver objects, which are used to invoke the receiver actions(). The Adapter pattern [3] can offer an interface between classes and objects. It requires that there are four classes Client, Target, Adapter, and Adaptee. Adapter must be a subclass of Target and must delegate Client calls to a method request() of the Target class to a method specificRequest() (with different interface) of the Adaptee class. To be able to do this, an Adapter instance needs an association to an Adaptee instance. If this structure occurs in the source code, current tools struggle to distinguish between the two patterns. An example of this confusion is shown in Figure 2. If these classes are matched to the pattern specification in Figure 1, there are two possible instances of the Command pattern, and two possible instances of the Adapter pattern (assuming that classes Receiver1 and Receiver2 are either Receiver or Adaptee classes). The rest of this paper introduces a lightweight, user-driven technique by which to single-out the correct pattern.

3 Detecting Patterns with User Knowledge

We propose a technique to enhance the detection process and eliminate false results. The proposed approach integrates user expert knowledge into the design pattern detection process. The approach operates by using conventional techniques to identify candidate patterns, and subsequently enabling the user to refine the list of candidates by supplying their individual knowledge to enenhance the behavourial design pattern detection results.
3.1 Integrating user knowledge
A schematic diagram of the approach is shown in Figure 3. This shows a structured user-driven process. It is divided into four phases; static analysis, dynamic analysis, constraint framework, and user knowledge phase.

Static Analysis
Generally, in traditional design pattern detection, it is common for the source code to be the main source of information for the analysed system. Accordingly, it is parsed to extract design facts in order to build a model of the system constraints. This phase produces an input file to be provided to constraint solver.

Constraint Framework
Starting from the fact that each design pattern has a particular abstract model that describes its structure, and each design pattern consists of one or more participant classes [3], we build a set of constraints. These constraints incorporate the classes, methods and the relationships between classes such as inheritance, association and method delegations.

Dynamic Analysis
This phase is intended to verify the design pattern candidates that were found in previous phase. For example, Eclipse debugger [8] can be used to run the program interactively while watching the source code and the variables during the execution. This makes it possible to trace and verify the execution and behavioural information of the of design pattern candidates.

User Knowledge
The final phase is the integration of the expert developer, to incorporate the sort of information that cannot be derived from static/dynamic analysis alone. Before integrating the user knowledge to the detection process, we have two assumptions related to the expert user. Firstly, assume the user has a good knowledge of the analysed system. Secondly, the user must be familiar with the design pattern structure. This phase receives the candidate patterns as input. The expert user looks at the result to start adding the hints by selecting the participated roles in the specified pattern structure and eliminating the unrelated roles. This process of automated analysis and user-driven input can be repeated in an iterative manner, thus gradually producing more accurate results.

3.2 Implementation
In the static analysis phase, the source code is parsed to extract design facts. Consequently, the INFAMIX parser [9] has been used. It exports the source code into .MSE file format, which is a generic format to describe the model, similar to XML. It contains eight entities; namespaces, packages, classes, methods, attributes, parameter and inheritance. Each of these entities has a unique identifier (e.g., (id: 3)) and it defines properties. The properties can be either primitive, like (name 'classA'), or they can point to another entity, like in the case of (parentPackage (ref: 5)) which denotes that the parent package property of ClassA points to the package with (id: 5). It is possible to refer to the entities using the ref: tag. An example of the file structure as follows:

(FAMIX.Class
(id: 3)
(parentPackage (ref: 5)))
To solve the detection problem as CSP, we need to implement it in a constraint programming language. Thus, we use the Z3py constraint solver [10]. In order to build a model for the constraint solver framework, we build a parser to retrieve all needed information from the .MSE file and build up the design facts (model). This model captures the design facts such as, relationships between classes and methods, inheritance, association, and method invocation.

Command Design Pattern Constraints:
- super-class (command)
- inh (concreteCommand, command)
- has-method (command, execute-method)
- abst-method (execute-method)
- override (concreteCommand, execute-method)
- invokes (invoker-method, execute-method)
- has-method (receiver, action-method)
- invokes (concreteCommand-method, action-method)

The constraint framework receives the extracted design facts as input, along with the design pattern constraints. It subsequently performs a matching process between the pattern constraints model and the extracted facts. It reports all structure that matches the abstract model. As a result of this step, the first design pattern candidates have been produced. In the dynamic analysis phase, we used the Eclipse debugger [8] to scrutinize program behaviour. This gives users deeper insight into the sequence of method calls from one class to another, expression and variable values, parameter passing. As a result, it helps to verify the pattern candidates by eliminating any class that does not participate and match the specified pattern behaviour (see example below). The output of this phase is a verified pattern candidate.

3.3 Example
To illustrate the proposed technique, we return to our motivating scenario in Figure 1. Having been processed by conventional constraint-based detection techniques (Steps 1 and 2 in Figure 3), the user is left with four candidate patterns: 2 Adapter pattern instances and 2 Command pattern instances. Looking at the Class names, the user might readily surmise that the patterns are Command patterns. This can be added as a hint by (for example) adding the following constraint, stating at the Acommand class plays the Command role:Command(Acommand) Having restricted the results to the Command pattern, the user might want to verify the results by dynamic analysis. Looking at the method invocation order, they might find to their surprise that Concrete2.m2() invokes Receiver1.action2() before ACommand.m2() is invoked. In other words, Concrete2 does not conform to the expected behaviour of the Command pattern and should be left out. Accordingly, this can be simply encoded as follows: Not(Command(Concrete2))

4 Evaluation
In this case study, the proposed technique has been applied to JHotDraw to Detect five behavioural design patterns. This case study investigates the involvement of the user expert in design pattern detection process, which aims to answer the following questions: 1) Does the tool expect the user to have good knowledge of the analysed system? 2) What are the types of user input are required? 3) How much amount of user input is required?

4.1 Methodology
JHotDraw v5.2 [11] is a drawing editor application framework for the creation of graphical editing applications. JHotDraw was developed to apply the design patterns concepts; this case study is based on v5.2, which is a publicly available mature framework. It was developed in Java and contains 9 packages, 148 classes, 1963 methods, and 17,819 lines of code. JHotDraw was chosen because it was built using well-known patterns. We focused on five behavioural design patterns that are especially problematic for conventional pattern techniques: Template Method, State, Strategy, Observer, and Command. As a baseline, we used no hints at all. The accuracy results (shown in terms of F-Score) are shown in Table 1. The resulting proposed patterns were used as the basis for the first iteration of hints. To avoid bias, we decided to focus our hints on generic knowledge about the patterns, as shown in Table 2. We avoided referring to the presence or absence of specific JHotDraw classes in a pattern. For all patterns, we provided the hints over two iterations. The results are shown in Table 1.

<table>
<thead>
<tr>
<th>Design Pattern No Hints Iteration 1 Iteration 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template Method</td>
</tr>
<tr>
<td>State/Strategy</td>
</tr>
<tr>
<td>Observer</td>
</tr>
<tr>
<td>Command</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1st Iteration</th>
<th>2nd Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template Method</td>
<td>30%</td>
<td>59%</td>
</tr>
<tr>
<td>State/Strategy</td>
<td>NA</td>
<td>27%</td>
</tr>
<tr>
<td>Observer</td>
<td>42%</td>
<td>47%</td>
</tr>
<tr>
<td>Command</td>
<td>14%</td>
<td>19%</td>
</tr>
</tbody>
</table>

TABLE 1. DETECTION RESULTS (F-SCORE)

4.2 Results and Discussion
Table 1 shows the results, where the result accuracy of the selected patterns has been refined during the detection process. The results indicate, just providing generic pattern-specific hints can lead to a substantial increase in accuracy. We have deliberately the use of implementation specific hints (e.g. class X is not a ConcreteCommand). Such hints would probably have increased the accuracy even further, and will form part of future work. This evaluation was intended as a proof of concept. As such, its results need to be interpreted with caution. Specifically, we have only evaluated the approach on a single system, with inputs from a single person (the principal authors), and with respect to only five patterns. It is possible that, on a different system, with a different user, and different patterns, the results would not look so favorable. We attempted to limit the bias by restricting the hints to non-implementation-specific facts.
5 RELATED WORK
Many studies have been made in the area of design pattern detection (e.g. [12-14]). Besides, many different recovery techniques used different recognitions methods for identifying patterns in source code. Some approaches have been considered. FUJABA [15] is one of the most popular design pattern detection tools. As it associates fuzzy values to pattern definition, and pattern recognition is driven by semi-automatic process. Ptidej [16] is an automated system to identify micro-structures similar to design patterns. It uses explanation based constraint programming to identify patterns in source code.

6 Conclusions and Future Work
This paper spotted the need to consider the pattern detection as an iterative user-driven process. We propose a framework to enable the user to supply domain knowledge that can complement the automated analyses, and produce more accurate results. Furthermore, our future work will look at a larger selection of systems, patterns, and consider inputs from a larger selection of user subjects.

7 References