Model-driven Detection of Design Patterns

Mario Luca Bernardi, Giuseppe Antonio Di Lucca
Department of Engineering, University of Sannio, Italy
{mlbernar,dilucca}@unisannio.it

Abstract—Tracing source code elements of an existing Object Oriented software system to the components of a Design Pattern is a key step in program comprehension or re-engineering. It helps, mainly for legacy systems, to discover the main design decisions and trade-offs that are often not documented. In this paper an approach is presented to automatically detect Design Patterns in existing Object Oriented systems by tracing system’s source code components to the roles they play in the Patterns. Design Patterns are modelled by high level structural Properties (e.g. inheritance, dependency, invocation, delegation, type nesting and membership relationships) that are checked, by source code parsing, against the system structure and components. The approach allows to detect also Pattern variants, defined by overriding the Pattern structural properties. The approach was applied to some open-source systems to validate it. Results on the detected patterns, discovered variants and on the overall quality of the approach are provided and discussed.

Keywords: Design Patterns, Software Comprehension, Software Maintenance, Software Evolution, Source Code Analysis

I. INTRODUCTION

In this paper we propose an approach to detect the Design Patterns (DPs) [1] implemented in an Object Oriented (OO) system by identifying the source components coding them. The approach defines a metamodel to represent a DP through a set of high level properties related to the source code elements, to the static relationships among them, and to their behavior. A system can be represented as an instance of this meta-model by a graph. The identification of the patterns is performed by a graph traversing algorithm that annotates the elements of the system type hierarchy with information on the roles they play in the patterns. A main advantage of the proposed approach over most of existing ones is that ours allows to identify variant forms of the classic DPs (as known in the literature). This is a particularly important issue since DPs are present in real world systems with many different variants [7], [6]. Our approach takes this into account and organizes the DPs catalog into a hierarchy of specifications, reusing existing ones. In particular a DP variant can be expressed as the modification of existing specifications by adding, removing or relaxing properties. Another advantage of using the proposed approach is that it allows to reduce size of the search space on the graph traversal. Indeed, since a Type can play several roles within several patterns at the same time, all classes should be considered. For this reason most of the approaches based on graph matching are inefficient, and performs very badly in detecting variants. In our approach, the analysis of a Type marks the code elements of the Type’s neighbour that are successfully bounded to patterns’ members. Existing bindings are then reused, when possible, during the traversal and provide chances to reduce the need of re-evaluating again properties for all Types, thus pruning the search space.

The approach we propose is not dependent on the pattern to mine and the mining process is based on declarative specifications. Hence, it is possible to write new pattern specifications derived from the existing ones (to detect variants) or to write them from scratch (to detect new patterns), with no impact on the mining algorithm. An eclipse-based tool, called Design Pattern Finder (DPF) has been developed to provide an automatic support to the approach.

The approach has been assessed by applying it to two open source java systems (JHotDraw and JUnit), based on DPs and considered also in several other studies [8], [7], [2]. The results have been validated against those from the analysis of an expert. The paper is structured as follows. Section II presents the meta-model defined to represent the DPs structure in terms of Properties. Section III presents the approach to find DPs in a Java system exploiting instances of the proposed meta-model. Section IV illustrates a case study conducted on two open-source Java systems. In Section V relevant related works are discussed and some comparisons with the proposed approach are made. Section VI contains conclusive remarks and briefly discusses future work.

II. A META-MODEL FOR DPS REPRESENTATION

The Figure 1 shows, as a UML class diagram, the meta-model defined to represent: the structure of an OO system (its Types and the structural relationships among them); the structure of the DPs (represented as a set of Properties modelling their structural elements), and the relationships among the DPs’ code elements and the Types. The structure of a system is modeled as a set of Types (i.e., Container, Value, Reference, and Array Types 1) where Reference Types are Interfaces and Classes and an Interface is implemented by one or more Classes. Reference Types are composed by Fields and Methods, and a Method can have Arguments. A ReferenceType can inherit from another ReferenceType as well as can contain another ReferenceType (e.g., an inner class). A Pattern is defined by the aggregation of the Properties characterising it. A Classifier Property, allows to introduce a Type (Class or Interface) used in a pattern specification (or to modify an already existing Type). A Classifier models a role needed by the pattern with respect

1 Array types are treated as separated types since they must specify the type of array’s components.
to its required internal structure and relationships with other Classifiers.

The **Data** Property is used to define a field in an existing Classifier (or override an existing field). The property can specify an existing Classifier as the field’s type or a compound type of an existing Classifier (like an array for a generic Collection). The **Behavioral** Property allows to define a method in an existing Classifier (or to override a method’s definition). The definition of the method includes the definition of its return type and its arguments and, optionally, of the method itself or for any of its arguments. This property can be used to define one or more of the required (or optional) behaviours of the Classifiers introduced in a pattern specification.

The **Dependency** Property describes the dependency between a pattern element (like a method) and another pattern element (as another method or a field). The **Invocation** Property models a call between methods already defined for some Classifiers in the pattern specification.

The **Delegation** Property specifies a mapping between a set of methods of a Class and a set of methods of an existing Classifier in the pattern specification. This allows to take into account the delegation for the patterns that require it.

The **Object Creation** Property models the creation constraints specifying the method or the field that needs the object creation and the Classifier of the created object. This happens for patterns expressing a mandatory object creation semantic as in the case of creational patterns but also for many patterns in the other categories.

The Figure 2 shows, as an example, a graph representing the instance of the meta-model for the pattern Observer. In the figure, a circle represents a Type, as specified by a Classifier property. Data properties are represented as hexagons (see the Observer’s fields connected to the ConcreteSubject). Methods and their arguments are rendered as diamonds and triangles respectively. The Property meta-classes are represented as stereotyped relationships. As shown in the figure, the Observer specification requires:

- a Field of type Container of AbstractObservers to be defined in the ConcreteSubject;
- a Delegation to be defined between ConcreteSubject and the Container type (by means of add and remove methods);
- the notify method to contain at least an invocation...
towards the update method of the AbstractObserver classifier;
• an object creation (to initialize the observer fields) in one of the constructors of the ConcreteSubject type.

This model expresses a quite common Observer (supporting only a single kind of event for each notify method) as implemented in several system. A variant to this specification could override the method Notify, in the Subject hierarchy, in order to add it a parameter to handle the multi-event Observer variant.

III. THE DETECTION PROCESS

The pattern mining process is structured as follows:
1) System source code analysis. The source code of the system under study is parsed, and the resulting AST is used as input to the meta-model instantiation.

2) Meta-model instantiation. A traversal of the system AST is performed to generate an instance of the system model, according to the defined meta-model. Rapid type analysis (RTA), class flattening and the inheriting of not public methods are used to build a system’s representation that can be used by the matching algorithm. RTA is used to handle late binding and hence the computed call graph reports a super-set of the real calls that can be executed at run-time.

3) Patterns’ models matching. The generated system model and the models of the DPs to be searched, stored in a repository, are detected by the matching algorithm.

Given a set of model graphs $P_1, \ldots, P_N$, representing the DPs specifications, and an input graph $S$, representing the analyzed system, mining design patterns means to find all the sub-graph isomorphisms from $P_i$ to $S$. The approach uses a matching algorithm inspired to the one proposed in [3] and modified to adapt it to the context of DPs’ mining. The algorithm we have defined allows to: (i) detect easily also patterns’ variants (handled as new pattern models inheriting and overriding properties of a parent pattern model), and (ii) reduce the search space by using constraints defined in the pattern specifications. Nodes’ and edges’ bindings are reused (across different pattern specifications) by marking traversed elements during the matching process, and the binding between already (and successfully) matched elements are no more considered (thus pruning the search space). The algorithm for each Classifier Property in the system evaluates matches with all Classifier properties of each pattern specification. A match between two properties is satisfied when properties are of the same kind and the nodes reachable from them can be matched. When a property $p_h \in P_i$ matches with a property $p_k \in S$, the nodes of properties reachable from both the pattern nodes and the system nodes are considered and new matches are scheduled in order to satisfy the originating match. When a match is satisfied, a binding between a pattern node and the system node is established. That binding is used to keep track of the elements of the system associated to patterns and the role they play in them. The algorithm continuously keeps a list of pending and successful matches for each pattern specification. The core of the algorithm used for performing the match is shown in Figure 3 (lines 1-6).

All Classifier properties of the system are considered, and for each one it tries to match a classifier properties of each pattern specification. The considered classifiers are dynamically updated during the traversal: each couple of properties for which a match already exists, or is pending, or has been unsuccessful (for a specific pattern), are removed from the list of remaining matches. The algorithm behaves in a different way depending on the kind of the properties that are matched. For a Classifier it will allow multiple successful matches across pattern specifications. Indeed, the same Classifier could be bound to more than one pattern, to handle multiple pattern instances involving the same classes (e.g. a Singleton that is also a Subject for an Observer). For the other properties (e.g. Data and Behavioral) a successful match will not allow those elements to be considered in future possible matches. This because if a Data or Behavioral element plays a role in a pattern, it could be imposed that the same element cannot be bounded to another role in another pattern (e.g. a method bound to the add method of a Component cannot be bound to the add of a Subject, thus the last match is not evaluated. This actually depends on the specification: for instance in the adapter case the adaptee methods could be already bounded to other patterns). This behavior ensures that the search space for all patterns specification is reduced by means of the constraints that are introduced at each successful match.

The match between two properties (line 5 of Figure 3, where $pp$ is a pattern property while $sp$ is a system property) is successful if the properties’ kind and attributes are congruent with each other, and matches for the properties nodes connected to $pp$ and $sp$ are successful in turn. The match algorithm evaluates each property connected to $pp$ with all the properties connected to $sp$ in order to find at least a successful binding for each mandatory pattern property. As showed in Figure 3 in lines 7-15 (reporting the case of Classifier properties) the match function proceeds in forward until there are no property nodes that can be

![Figure 3. A sketch of the detection algorithm.](image-url)
considered. For every property node found on the pattern model, a new match is tried for each system node. When the forward phase is completed and it was successful, a backward matching step is tried: in this case all properties having a reference to the pattern and to the system are considered to be matched against each other. For a match to be successful, both forward and backward steps must result in at least a binding (line 14). When the forward step fails, the backward step is not executed at all and the match is unsuccessful. Moreover existing bindings are reused while pending matches, suspended during the evaluation of matches for the properties in the neighbourhood, are considered as satisfied since they are part of the ongoing binding.

IV. CASE STUDY

The proposed approach has been validated on two open source Java software systems: JHotDraw 6, a Graphical Editor framework (made up of 25,308 LOC and 83 classes), and JUnit 3.7, a unit testing framework for Java programs (made up of 9,743 LOC and 27 classes). We choose these systems because their development is explicitly based on design patterns and hence are good at evaluating design pattern mining approaches. Moreover, they have been extensively studied in literature and hence benchmark already exists for these systems. This means that it is easier to evaluate precision and recall providing quantitative data on the quality of detection.

To assess the effectiveness and the correctness of the approach its results were compared with the ones indicated by an expert that validated the DPs identified by the tool by analysing the systems’ code and documentation, and the results from other works known in the literature. The Table I, for each of the two analysed systems, reports: the name of the DPs searched in the code (column 'Pattern'); the number of each searched Pattern detected by the proposed approach (column 'Detected'); the number of each searched Pattern the experts said to exist actually in the system (column 'True'); the number of False Positive (column 'FP'), i.e. the number of DPs instances detected by the tool but not validated by the expert; the values of precision and recall for the results by the tool (columns 'Precision' and 'Recall' respectively).

Some false negatives, i.e. pattern instances that exists in the system (as specified by the expert) but missed by the detection process, were found along the validation process.

As clarified below, for some of these pattern, variants are declared in a pattern specifications repository accessible by the matching algorithm.

Patterns like Command, Composite or Observer but also Visitor (that is based on double dispatch) are better identified since their specification include both static and behavioral relationships. Indeed they have a lower number of false positives than those patterns with a less constrained structure or with limited or absent behavioral properties. The false negatives were related to patterns that were implemented by a not standard way. Anyway they were in a lower number than false positives. This is also related to the variants. False negatives occurred for State, Singleton and Prototype Dps in JHotDraw, and for State and Observer Dps in JUnit.

The proposed approach can help to distinguish among patterns that have same static structure but different behaviours. For example, in order to distinguish Command from Adapter (the object version), the approach uses information inside the invocation property requiring that the Execute method, in the concrete subclass, is implemented by invoking a method of a class still bound to a Command. The same is for Composite/Decorator where the Decorator is required to specify a delegation towards the decorated object.

<table>
<thead>
<tr>
<th>Pattern Variants</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JHotDraw 6</td>
</tr>
<tr>
<td>Composite A</td>
<td>x</td>
</tr>
<tr>
<td>Composite B</td>
<td>-</td>
</tr>
<tr>
<td>Observer A</td>
<td>x</td>
</tr>
<tr>
<td>Observer B</td>
<td>-</td>
</tr>
<tr>
<td>Observer C</td>
<td>-</td>
</tr>
<tr>
<td>Singleton A</td>
<td>x</td>
</tr>
<tr>
<td>Singleton B</td>
<td>-</td>
</tr>
<tr>
<td>Singleton C</td>
<td>-</td>
</tr>
</tbody>
</table>

Table II: Summary of Identified Design Patterns Composition and Variants.

For what concerns the defined DP variants, the Table II shows the ones contained in the pattern repository. In particular, the repository contains three specifications for the Observer pattern: the first one (A) uses the Java types (Observable class and Listener interface) while the other uses a generic interface to be found within system classes and the third one defines a multi-event observer in which type of event is passed to the notify method. For the Composite pattern only two versions are contained in the repository: the classic proposed in literature by Gamma [1] (version A) and a version in which an intermediate abstract class implementing the core method for components has been inserted in the Component hierarchy (version B). For the Singleton DP, 3 variants are provided: a first version (A) with a static getter and a private constructor; a second one (B) taking concurrency into account and a third version (C) in which there is single instance for each object identifier passed to the singleton itself.

We’ve measured the times of running the approach for each step of the detection process. The patterns matching step is the most CPU time consuming. We cannot show detection times for each pattern since our approach uses...
the successful identifications across pattern specifications as constraints to improve the performance and hence the detection times are dependent. However, we calculated the average time to detect a single pattern (538 ms for JUnit and 2,208 ms for JHotDraw) and it resulted to be comparable to the one of other approaches. Moreover, experiments performed tuning the patterns specifications showed that performances can be considerably improved by identifying structural and behavioral constraints that are effective and able to select a well defined variant of a pattern. Hence our approach is more effective when specifications are structured in a hierarchy and each specification is dedicated to a patterns variant.

V. RELATED WORK

The problem of mining DPs in existing OO systems has been faced and discussed in several works, and different methods and techniques have been proposed to support it. In [4], De Lucia et al. present some case studies of recovering structural design patterns from OO source code. They use a recovery technique based on the parsing of visual languages, and supported by a visual environment automatically produced by a grammar based visual environment generator. In [8], [2], Guheneuc et al. propose an approach to semi-automatically identify micro-architectures that are similar to design motifs in source code and to ensure the traceability of these micro-architectures between implementation and design.

A design pattern detection methodology based on similarity scoring between graph vertexes is proposed in [7]; the approach is able to also recognize patterns that are modified from their standard representation. The approach exploits the fact that patterns reside in one or more inheritance hierarchies (in order to reduce the size of the graphs to which the algorithm is applied).

VI. CONCLUSIONS AND FUTURE WORK

A method to identify DP in existing OO systems has been presented. A meta-model has been defined to represent DPs by a set of Property specifying each DP, and the system to mine. The identification of DPs is carried out by performing an algorithm that matches the models of the DPs against the model of the system to detect those components cooperating in a way that satisfies a pattern model. The approach also allows to identify DPs variants as modification to pattern specification already defined. The approach has been applied to two Java systems producing good results. Future work will be devoted to extend the catalog of identified patterns and to perform further experimentation on a wider set of open source systems.

REFERENCES


