Design Pattern Finder : A Model-driven
Graph-Matching Approach to Design Pattern Mining

Mario L. Bernardi¹, Marta Cimitile², Giuseppe Di Lucca¹

¹Dept. of Engineering-RCOST, University of Sannio, Benevento, Italy
²Fac. of Jurispudence, Unitelma Sapienza University, Italy
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 work 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
Contents

1 Introduction .................................................. 3

2 State of the art .................................................. 5
   2.1 Type of analysis ........................................... 5
   2.2 Searching Methods ....................................... 6
   2.3 Comparison with the proposed approach .................. 7

3 The proposed approach ........................................ 8
   3.1 A Meta-model and a DSL for Design Pattern Representation .................................................. 8
   3.2 The Detection Approach .................................. 10
       3.2.1 Design Patterns Graphs .............................. 11

4 Case Study ..................................................... 16
   4.1 The Pattern Library ..................................... 16
       4.1.1 Adapter .................................................. 17
       4.1.2 Bridge ................................................... 17
       4.1.3 Command ............................................... 18
       4.1.4 Composite .............................................. 18
       4.1.5 Observer ............................................... 19
   4.2 Evaluation strategy ..................................... 20
   4.3 Selected systems and searched Patterns .................. 21

5 Discussion of Results ........................................ 23
   5.1 Quantitative Analysis - Precision and Recall .............. 23
   5.2 Performance issues ...................................... 25
   5.3 Threats to Validity ....................................... 26

6 Conclusions and Future Work ................................ 30
1 Introduction

A Design Pattern (DP) represents an appropriate solution specified for a recurrent design problem.

Gamma et al. in [7] listed the first catalog of DPs. In the last years is diffused the idea that detection of the instances of Design Patterns (DPs) from Object Oriented (OO) software systems can help to better structure, understand, maintain and reuse them [14]. However, the lack of adequate documentation in a software system, may make hard to understand which are the adopted design solutions and patterns and where (in which code components) they are implemented. In this paper we propose an approach to detect the DPs [7] implemented in an Object Oriented system by identifying the source components coding them. The matching approach is centered on a metamodel representing both the software system and the searched DP through a set of high level properties related to the source code elements, the static relationships among them, and their behavior. A system is represented as an instance of this meta-model (modeled as a large graph of elements and properties about them). 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 each pattern model. An advantage of the proposed approach over most of existing ones is that it also allows to easily specify variant forms of the classic DPs (as coded in the literature). This is an important issue to address since it is well known that DPs are present in real world systems with many different variants [26, 29]. Our approach takes this into account and organizes the models of DPs to be identified as a hierarchy of declarative specifications. In particular a DP variant can be expressed as a set of changes to an existing specification by adding, removing or relaxing properties. Hence, it is possible to write a new pattern specification deriving it from an existing one (to detect a variant) or to write it from scratch (to detect new kind of pattern), with no impact on the mining algorithm. This approach also helps to reduce the size of the search space since variants share part of the search tree.

An eclipse-based tool, called Design Pattern Finder (DPF) was developed to provide an automatic support to the approach.

The approach has been assessed by applying it to four open source java systems (JHotDraw 6, JUnit 3, JHotDraw 7, Apache Avro 1.6), based on DPs and considered also in several other studies [30, 26, 10]. For the latter two systems we use also Tsantalis DPs detection tool [20] to compare the proposed approach with a similar one. The results have been validated by the analysis of an expert.

The remainder of this report is structured as follows: chapter 2 shows relevant related works and some comparisons with the proposed approach are made. Based on these, chapter 3 presents the meta-model defined to represent the DPs structure in terms of...
Properties and the approach to find DPs in a Java system exploiting instances of the proposed meta-model. Chapter 4 illustrates a case study conducted on 4 open-source Java systems and is followed by the discussion of the obtained results (chapter 5). Finally, chapter 6 contains conclusive remarks and briefly discusses future work.
2 State of the art

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. According to this, some reviews on current techniques and tools for discovering architecture and design patterns from OO systems, are provided in [5] and [22]. In the last work, authors classified pattern recovery techniques basing on the type of analysis used and the searching methodology adopted.

2.1 Type of analysis

We can consider different kind of pattern recovery approaches: structural analysis, behavioral analysis, semantic analysis and formal specification/composition analysis to recover patterns from the source code of legacy applications. Structural analysis approaches consist recovering the structural relationships from available source code artifacts. They focus on recovering structural design patterns such as Adapter, Proxy, Decorator, etc. These approaches consider inter-class relationships to identify patterns structural properties. An example of structural analysis approaches is [27]. In this work the source code is parsed using a third party commercial tool called Understand for C++. The tool extracts the entities and the references from C++ source code and stores them in a database. Queries are performed on the database to extract different properties of patterns. In their experimentation authors recovered Singleton, Factory, Template, Observer and Decorator patterns from a VCS (Version Control System). The adopted VCS contained only 125 classes that are not available on the web.

Behavioral analysis approaches adopt dynamic analysis, machine learning and static program analysis techniques for patterns behavioral aspects extraction. They can be used together with structural analysis when patterns are structurally identical or have a weak structure (for example, State and Strategy patterns are structurally identical). The limit of these approaches is that they give many false positives at the increasing of the number of execution traces [6].

Semantic analysis approaches complete the structural and behavioral analysis reducing the false positive rate for recognition of different patterns. They use naming conventions and annotations which contain the role information about the classes and methods [6], [21]. This analysis can be used for recovery of patterns having similar static and behavior properties (i.e. Bridge and Strategy patterns). Different techniques are used for semantic analysis. In [6], three options are discussed for semantic analysis and they conclude that naming conventions are most appropriate and feasible option.

Finally, formal specification/composition analysis of design patterns includes some ap-
approaches extracting patterns from source code based on formal specifications of design patterns. It is important to supplement different pattern detection approaches by formally specifying different patterns [2], [8], [29]. Moreover, design patterns have different implementation variants and any formal specification of patterns can help to specify the possible variations in different patterns as well as overcome the challenges of capturing the semantics for patterns. These approaches use formal specification languages to specify different design patterns and most of the specification languages have tool validating the specifications correctness and completeness [8].

2.2 Searching Methods

We can distinguish different methods for patterns recovery: Database queries, Constraint resolver, Metrics, XPG formalism and parsing, UML structures and matrices, miscellaneous approaches.

Several pattern recovery techniques use Database queries [21], [24], [15] for extracting patterns. They give an intermediate representation of the source code (i.e. ASG, AST, XMI, metadata and UML structures) and then use SQL queries to extract pattern related information.

The queries performances depend by the used database can be scaled very well, but such queries are limited to the information available in the intermediate representations. So the limit of this approaches is that there are no intermediate representation format storing all the information in source code. Moreover, they are used for structural and creational design patterns and they only partially support behavioral design pattern recovery.

Several tool suits and frameworks are used to identify idioms, macro-patterns, design patterns and design defects with explanation based constraints programming techniques. For example, in [12], authors recover patterns using a multilayered approach which focuses on ensuring a optimal recall rate, but precision is scarified and performance is low.

Metric based techniques compute program related metrics (i.e. generalizations, aggregations, associations, interface hierarchies) from different source code representations and compare their values with source code design pattern metrics. These techniques [28], [14], [1] are computationally efficient because they reduce search space through filtration [11]. The limit is that they were experimented on a few number of patterns. Moreover, their precision and recall is low.

XPG formalism and parsing techniques use SVG (scalable vector graphics) format for the intermediate representation of the source code and represent design patterns in a visual language by mapping the visual language grammar of each pattern with the graph representation. They give a precise visualization but are limited only to structural design patterns. Moreover, in our knowledge, the existing experimentation are limited to few patterns [17] and not show any recall rates.

UML structures and matrices techniques [26], [1], [19] represent structural and behavioral information of software systems. They apply different techniques to match the
design pattern template metrics with the matrices generated for the system. In particular, a design pattern detection methodology based on similarity scoring between graph vertexes is proposed in [26]. The approach is able to also recognize patterns that are modified from their standard representation. It 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).

These approaches are computationally efficient and have good precision and recall rates. Their limit is that they miss to extract the implementation variants of similar design patterns. Furthermore, they are only limited to a few number of patterns.

Finally, there are some well known techniques that can be classified in the above categories (i.e. Fuzzy reasoning, Bit vector, Data flow and control flow, Minimum key structure, Predicate calculus, rho calculus, Runtime analysis, Formal Semantic, XML matching, REQL query, Machine learning, Structural and behavioral parsing, Static Reference Flow analysis, Concept Analysys, Data Mining). For example, in [17], 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.

According to these techniques, the research studies are focused on the formalization of empirical evaluation criteria [6], [17], [25]. In fact, each applied technique should be evaluated on the basis of certain criteria and different authors gave taxonomies and/or frameworks for the evaluation of techniques used in the area of reverse engineering.

2.3 Comparison with the proposed approach

A main advantage of the proposed approach over most of existing ones is that it 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 [26, 23]. 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 is that the proposed approach is based on a low level system metamodel that is able to represent elements down to statements and expressions. This allows to consider and reason also on behavioral properties that can be used (i) to improve search state reduction; (ii) to distinguish between patterns that have the same structure but different behaviour [26]. This allows to perform a finer grain detection separately identifying patterns that have the same structure but used (and behave) in a different way.
3 The proposed approach

3.1 A Meta-model and a DSL for Design Pattern Representation

The mining approach is centered on a meta-model used to define a Domain Specific Language (DSL) suitable to describe pattern mining tasks. The DSL represents relevant properties of OO software systems. Structural relationships (inheritance, implementation, type nesting and visibility) among the Types and the relationships among code elements are considered. Both design patterns and the system under study are represented as a set of Properties modelling both the structural and behavioral elements. The Figure 3.1 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.

As Fig. 3.1 shows, the structure of a system is modeled as a set of Types (i.e., Container, Value, Reference, and Compound Types) along with their relationships. Reference Types are in turn composed by Fields and Methods, and a Method can have zero or

---

1 Compound types are treated as separated types since they must specify the base type of compound. In this class are also arrays and generic types.
more Arguments. A ReferenceType can of course inherit from another ReferenceType as well as can contain another ReferenceType (e.g. an inner class).

A Design Pattern in this model is defined by the aggregation of several 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). Also, it allows to define constraints, if any, on its super-type or its implemented interfaces. A Classifier models a role needed by the pattern with respect 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 of Classifiers defined 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.

From the meta-model of Fig. 3.1 we derived a DSL language to express pattern specification with the following goals in mind:

- easy to write by hand
- capable to represent constraints on code element
- reusable pattern specifications
- support of pattern variants detection

There is a long tradition in the programming language community of domain-specific languages tailored to a particular application domain. In the context of pattern mining two characteristics that strongly point towards using a domain-specific language are: (i), the structure of software system (and patterns), in which modules, statements and expression can be composed using recursive rules; and (ii), the need (and difficulty) of succinctly representing the constraints on such structure. Our DSL language takes inspiration from many languages and systems proposed in the past. Namely the SDF, Crocopat and Grok [13, 12, 14].
Figure 3.2: An Example of DSL instance: the Observer Pattern specification

The Figure 3.2 shows, as an example, the specification written in the proposed DSL to detect the Observer pattern. The core part of each specification resides in the \textit{find type} declaration which defines the node roles needed to express the pattern along with constraints on their multiplicity.

\section*{3.2 The Detection Approach}

The pattern mining process, sketched in Figure 3.3, comprises the following steps:

- **Definition of the patterns specifications repository**, written using the DSL based upon the proposed metamodel (step 1).

- **Pattern Models instantiation**, in which parsing of the pattern models (written using the proposed DSL) is performed to generate instances of the design pattern graphs used to perform the match (execution of the DSL2GRAPH block, step 2).
Figure 3.3: A sketch of the overall design pattern mining process.

- **System source code analysis**, in which the source and byte code of the system under study are parsed and the complete ASTs of the system are produced (execution of the step 3).

- **System Model instantiation**. A traversal of the system AST is performed to generate an instance of the system model, conforming to the defined meta-model. Rapid type analysis (RTA), class flattening and inlining of not public methods are exploited in order to build a system’s representation suitable for the matching algorithm\(^2\). This is accomplished in step 4 by executing the AST2GRAPH block.

- **Design Patterns Matching**. This is the main task (step 5) in which the generated system and patterns graphs are detected by the matching algorithm.

In the remaining of the section, the overall detection process is described in detail.

### 3.2.1 Design Patterns Graphs

In this section some background definitions are provided to make the following discussion self-contained. A Design Pattern graph (DPG) is the building block of our search in the system graph and is used to identify sub-graphs of interest (the design pattern instances) occurring in the system graph. Essentially, a DPG can be regarded as an attributed graph specifying a set of predicates on the attributes that must hold. Formally:

**Definition 1.** *Design Pattern Graph* — A design pattern graph is a pair \( DP = (P, AC) \), where \( P \) is an attributed graph and \( AC \) is a set of predicates on the attributes.

\(^2\) Note that 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. This however only lower the precision in very few cases. A discussion on the impact on the detection quality is however reported in threats to validity section.
The predicate set AC contains compound expression made of conditions on nodes, edges and attributes of P. To express design pattern graphs the proposed DSL, derived from metamodel in Figure 3.1, can be used. The Figure 3.2 shows a small design pattern graph to search for a simple Observer. Compound predicates can be broken down to simple predicates on individual (or set of) nodes or edges, like the right side of the figure shows.

We introduce at this point the notion of design pattern graph matching which generalizes sub-graph isomorphism with evaluation of the predicates on the attributes.

**Definition 2.** Graph Pattern Matching — A design pattern graph DP(P, AC) is matched with a system graph S if there exists an injective mapping \( \phi : V(P) \rightarrow V(S) \) such that (i) For \( \forall (u, v) \in E(P), (\phi(u), \phi(v)) \) is an edge in S, and (ii) predicate \( AC_\phi(S) \) holds.

If a design pattern graph is matched to a system graph, the binding between them can be used to access the sub-graph on the system (either the sub-graph structure or attributes and properties on nodes and edges). We define a matched graph to denote the binding between a design pattern graph and the system graph.

**Definition 3.** Matched Design Pattern Graph — Given an injective mapping \( \phi \) between a pattern DP and a graph S, a matched graph is a triple \( \langle \phi, DP, S \rangle \) and is denoted by \( \phi_{DP}(S) \).

The Figure 3.3 outlines the detection algorithm. The specification expressed as a design pattern graph DP is rewritten by means of a set of predicates on individual nodes \( AC_u \) and edges \( AC_e \). For each node u in the pattern DP, there is a set of candidate matched nodes in S for which the constraints \( AC_u \) hold. These nodes define a (partial) matched design pattern graph referred as candidate neighborhood of node u and denoted by \( \phi(u) \):

**Definition 4.** Candidate Neighborhood — The candidate neighborhood \( \phi(u) \) of node u is the set of nodes in graph S that satisfies the predicate \( AC_u \):

\[
\phi(u) = \{ v \mid v \in V(S), AC_u(v) = true \}
\]

Hence the search space of our design pattern graph matching for a pattern DP and system S is defined by the candidate neighborhoods of all nodes belonging to the patterns specification as follows:

**Definition 5.** Search Space — The search space of design pattern graph matching of pattern graph DP on system graph S is defined as the product of candidate neighborhood for each node of DP:

\[
\phi(u_1), \times \ldots \times, \phi(u_k) \in S,
\]

where \( u_1, \ldots, u_k \in DP \).

The algorithm, in two phases, starts at line 4 by calling (line 6) the forwardNeighborhoodAnalysis function (lines 13-21) which computes the candidate neighborhood for
List<MatchedGraph> instances = ...;

void start()
begin
forwardNeighborhoodAnalysis(DP)
for i = 1 to k do
  Match(i);
end
end

void forwardNeighborhoodAnalysis(V)
begin
foreach node u V (DP) do
  Phi(u) = { v in V (S) | AC_u (v) = true }
  (1) computation Phi(u)
  (2) Reduce Phi(u1) ... Phi(uk) using lookahead and properties constraint
end
end

void Match(i)
begin
foreach v in Phi(u_i) | v is free do
  if not checkNeighborhoodBindings(ui , v)
    then continue;
  Phi(u_i) = v;
  if i < |V(DP)| then Match(i + 1);
  else
    if AC_Phi (S) then instances.add(Phi());
end

boolean checkNeighborhoodBindings(ui , v)
begin
  foreach edge e(u_i , u_j ) in E(DP), j < i do
    if (edge e_1(v,Phi(u_j)) not in E(S))
      or (not AC_e(e_1)) then return false;
  end
  return true;
end

Figure 3.4: A sketch of the detection algorithm.
each node $u_k$ in the design pattern graph. This function ends after performing a pruning step of the search space (better described in the following).

The second phase (on lines 23-40) performs a search, over the product $\phi(u_1) \times \ldots \times \phi(u_k)$ using a depth-first traversal, to find a sub-graph isomorphisms. The Match(i) function iterates on the $i_{th}$ node to find valid bindings for that node. Procedure check-NeighborhoodBindings$(u_i, v)$ examines if $u_i$ can be mapped to $v$ by considering their edges and attributes. Line 28 maps the node $u_i$ to $v$. Lines 29-32 continue to search for
the next node or, if it is the last node, evaluate the predicate AC to check constraints. If it is true, then a valid binding \( \phi : V(DP) \to V(S) \) has been found and is added to the list (line 31). Since the worst-case complexity of the matching algorithm is \( O(n^k) \), where \( n = |S| \) and \( k = |P| \), to make the algorithm usable on real systems, a search space reduction technique must be used. Our approach uses system and pattern information to reduce the size of candidate neighborhoods and exploits a look-ahead requiring, for each node \( u_i \) of the design pattern graph, a valid (partial) binding of the neighborhood sub-graph centered in \( u_i \) and having a fixed distance \( r \) from it.

For each candidate neighborhoods, structural information (e.g. nodes and edges) and predicates on attributes (types and properties of nodes and edges) are used to prune matches that would not produce acceptable solutions. This neighborhood knowledge can be exploited to prune infeasible sub-graph at an early stage and obtain a reduced set of candidates on which to perform the full depth first matching (that is resource- and time- expensive). There is a trade-off with respect to how candidate neighborhood sub-graphs are built. They increase pruning power as the look-ahead increases, but their construction is of polynomial complexity (with respect to look-ahead). Our actual implementation uses a look-ahead equals to 1 (immediate neighborhood) and we found that with a look-ahead equals to 2, even if for some patterns (those that have a rich structure or highly constrained) time was greatly reduced, for others the more complex neighborhood construction increased the total time (the average time remained almost the same).

To exemplify the process, Figure 3.5 reports a simple running example. It shows the candidate neighborhood analysis related to the pattern specification in Figure 3.2, performed on few nodes of the pattern graph, and the resulting bindings. For each pair of nodes \( u_i \in V(DP) \) and \( v_j \in V(S) \), the neighborhood sub-graph of \( u_i \) and \( v_j \) are matched to find candidate neighborhood. The Figure 3.5 just reports few interesting cases. In the step 1 the pair considered is \( (AS, Figure) \) and hence the immediate neighborhoods of respectively DP and S are considered. This match fails since pattern node neighborhood is not a sub-graph of S. By converse the step 2 on pair \( (AS, Figure) \) is a successful match since DP neighborhood is congruent with S and all the constraints are satisfied (nodes and edges are of the same types, and multiplicity constraints are honored). Hence several conditioned bindings are established for the matched candidate neighborhood. The step h is for the pair \( (Figure, AO) \). Due to structural differences this match correctly fails avoiding an infeasible candidate neighborhood (since the Figure is an AbstractSubject). This because the pattern structure of the observer pattern has a quite good pruning power. Simpler patterns may generate a higher number of candidate neighborhoods that must be taken into account in the second phase of the algorithm in which the full depth-first matching performed increases time and space requirements. Hence, to further reduce search space, the algorithm is executed on all the design pattern graphs in the specification repository and the previous bindings are taken into account when performing the subsequent candidate neighborhood analysis. When a pattern model does not allow multiple bindings for a pattern element, existing established bindings of already analyzed patterns are used as further constraints to improve the search space reduction.
4 Case Study

To perform the validation, a prototype tool called Design Pattern Finder (DPF) has been developed. It was implemented as a set of Eclipse plugins based upon JDT (to extract information on the systems’ static structure down to code statement level), and upon the EMF framework (to implement the meta-model). In the way to validate the proposed approach, we decide to adopt it for pattern analysis of 4 java software systems. We evaluated effectiveness and the correctness of 2 of the systems considering the DPS identified by an expert that analyzed the systems’ code and documentation. For the other 2 systems, we evaluate precision and recall comparing the obtained results with the ones obtained using another approach and tool that is very known in the literature \[26\]. For both the tool results the expert validated the correctness of the obtained DPs.

In this way we can generalize the obtained results and we have also a comparison of our tool with another tool that is very diffused in the community.

4.1 The Pattern Library

At the time of writing the following patterns specification are in use in DPF:

- Adapter Class
- Adapter Object
- Bridge
- Command
- Composite
- Decorator
- Factory Method
- Observer 1st - Simple
- Observer 2nd - Multiple event
- Prototype
- Proxy
- State
To better highlight the DSL structure and capabilities, a discussion clarifying the structure of some of the most relevant mining specifications is presented in the remaining of the section.

### 4.1.1 Adapter

```plaintext
pattern Adapter {
  find type X(1), Y(1) where {
    X has field f of-type Y
    X has methods—set adaptationSet each {
      delegates—to f
    }
  }
  variant "ADAPTER CLASS" {
    X inherits—from Y
    element—override methods—set adaptationSet each {
      delegates—to u in methods—set(Y)
    }
  }
}
```

The Adapter design pattern exists in two forms. The class Adapter is based on inheritance (delegates to inherited methods) while the object Adapter exploits composition. In the specification of the object adapter, two template types are searched: an adapter role X incapsulating a reference to an adaptee type Y.

X must define a field of type Y and should implement the methods of the X interface in terms of methods of the Y one (referring to the adaptationSet method-set defined in the specification, each method of the set must delegate to a method of f field). The class adapter instead, is specified as a variant of the object adapter; in this case X is forced to inherit from Y and adaptationSet is built considering the methods inherited from Y.

### 4.1.2 Bridge

The Bridge pattern specification requires to find four roles: a single Abstraction (A) linked to a single AbstractImplementor. Several concrete subclasses of these two roles are allowed (RefinedAbstraction (R) and Implementor (I)). Hence each type of the system
pattern Bridge \{
  find type A(1), R(*), AI(1), I(*) where \{
    AI has methods—set abstractionSet
    I inherits—from AI
    R has field ai of—type AI
    R overrides methods—set abstractionSet each \{
      delegates—to ai
    }
  }
\}

pattern Command \{
  find type AC(1), CC(*), R(*) where \{
    R has methods—set receiverSet
    AC has method e
    CC inherits—from AC
    CC has field r of—type R
    CC overrides e \{
      delegates—to r in receiverSet
    }
  }
\}

bound to R should implement the type bound to A while each type of the system bound to I should implement the type bound to AI. Moreover each R has a field of AI and a subset of his method should delegate to AI methods (each refined abstraction is implemented in terms of the AbstractImplementor interface).

4.1.3 Command

The Command pattern requires three template types. A single AbstractCommand (AC) inherited by several ConcreteCommands (CC) and a set of Receivers (R). Each Concrete Command has a reference to a receiver and overrides the execution method of the Abstract command (the e method in the specification). This method delegates to a method in the set “receiverSet” of the Receiver.

4.1.4 Composite

The composite specification models the Component hierarchy using three roles: Leaf (LF), Component (C) and CM (Composite). Both Leaf (LF) and Composite (CM) must inherit the Component role. While LF has inner structure (no reference to inner components), CM must own a reference to a collection (named “cm” in the specifica-
pattern Composite {
  find type \^C(1),CM(*),LF(*) where {
    \^C has method a{
      has return type void
      has param c of\-type \^C
    }
    \^C has method r {
      has return type void
      has param c of\-type \^C
    }
    \^C has methods\-set componentSet
    LF inherits\-from \^C
    LF has\-not container co of\-type \^C
    CM has container cm of\-type \^C
    CM inherits\-from \^C
    CM has methods\-set componentSet each {
      delegates\-to cm
    }
  }
}

operation) having C as base type. Each method of the composite CM(referred to as “componentSet”) delegates to container methods (to allow adding/removing/iterating over inner components).

4.1.5 Observer

The 1st Observer specification requires:

- a single AbstractObserver (AO) and several ConcreteObserver (CO);
- a single AbstractSubject (AS) and several ConcreteSubject (CS);
- a container of AbstractObservers to be defined in the ConcreteSubject (the field “o”);
- the methods a and r (that play roles of add and remove) to be defined in the AbstractSubject and overridden in ConcreteSubjects
- a Delegation to be defined between a and r of ConcreteSubject and the add/remove methods Container type;
pattern Observer\_1st {
    find type S(*), AS(1), O(*), AO(1) where {
        O inherits from AO
        O has methods set updateSet
        S inherits from AS
        S has container o of type O
        S has method a {
            delegates to o
        }
        S has method r {
            delegates to o
        }
        S has methods set notifySet each {
            delegates to o
            calls u in O.updateSet
        }
    }
}

- notify methods set (called "notifySet"); each method of the set must contain an invocation towards the update method of the AbstractObserver classifier;

- an object creation (to initialize the field "o") in the constructor of the Concrete-Subject type.

This model expresses a quite common Observer (supporting only a single kind of event for each notify method) as usually implemented in several systems. A variant to this specification overrides the set "notifySet" and the method "u" to represent the multi-event Observer and is showed in 5.1.

### 4.2 Evaluation strategy

Given the big number of approaches that were proposed in the last years, it became very important to evaluate our approach with some criteria that are diffused in literature and that permit to compare the obtained results with results deriving by other existing approaches and tools. According to [22], design pattern recovery techniques can be evaluated basing on different features, but in order to assess effectiveness and correctness we decide to use the concept of precision and recall [20]. The motivation for evaluating precision and recall stems from the disparity of results.

Moreover, in the way to generalize our results, we decide to consider 4 different systems. For 2 of these systems, to assess the effectiveness and the correctness of the approach, the 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. For the other 2 systems we compare the obtained results obtained with the ones obtained using another tool and an expert evaluated for each tool the obtained results.

In order to compute recall and precision we assume that a pattern instance can be classified into one of four categories:

- true-positive ($T_P$: correctly found),
- false-positive ($F_P$: incorrectly found),
- true-negative ($T_N$: incorrectly missed),
- false-negative ($F_N$: correctly missed).

On that base, precision is defined as the ratio of correctly found occurrences to occurrences provided by the tool and is given by:

$$ Precision = \frac{T_P}{T_P + F_P} \quad (4.1) $$

Recall is the ratio of correctly found occurrences to all correct occurrences and is given by:

$$ Recall = \frac{T_P}{T_P + F_N}. \quad (4.2) $$

The Gold Standard (GS) used as reference is the set of all true positive instances. As said it was computed in a different way for the two systems groups.

For JUnit and JHotDraw 6, it was calculated by an expert performing a complete source code inspection searching patterns by hand. Hence it could be assumed to be correct and complete (it is still exposed to bias and human mistakes but, due the size of the two systems, these effects are limited).

For JHotDraw 7 and Apache Avro 1.6, it was computed using the correct results produced by both DPF and Tsantalis tools. Hence it could lack pattern instances missed by both tools but allows a direct comparison.

A perfect precision score of 1 means that every occurrence found was correct (but says nothing about whether all correct occurrences were found), whereas a perfect recall score of 1 means that all correct occurrences were found (but says nothing about how many incorrect occurrences were also found). Both measures are therefore complementary.

### 4.3 Selected systems and searched Patterns

We decide to consider 4 open source java software systems of increasing sizes (namely JUnit 3.7, JHotDraw 6, JHotDraw 7 and Apache Avro 1.6). The DPs that are considered in this investigation are the most common ones in literature: Factory Method, Prototype, Singleton, Adapter, State, Strategy, Composite, Decorator, Observer, Template Method, Command, Proxy, Visitor (in tables 5.1 and 6.2 only the DPs for which we found at least one occurrence are reported).
To clarify the context of the analysis, the characteristics of these systems are reported in Table 4.1. For the former two systems, to assess the effectiveness and the correctness of the approach, the results were compared with the ones reported by an expert that manually inspected source code. For the latter two systems, since their sizes did not allow a complete manual code inspection, the assessment was performed comparing results obtained by DPF with results obtained by using another tool (the one proposed by Tsantalis et. al. in [26]). An expert evaluated, for each tool, the obtained results.

Systems were chosen in order to generalize our previous preliminary investigation results reported in [3]. For this reason we choose systems explicitly based on design patterns (like JHotDraw 6 and 7) and two real world industrial systems (JUnit 3.8 and Apache Avro 1.6). Moreover the system were chosen with increasing sizes to evaluate the scalability of the algorithm and to validate the quality of results on a larger code base.

The table shows that the selected systems are very different for dimension and some of them have different objectives or are selected in different versions. We choose the systems in this way with the aim to generalize our investigation results. Moreover, these systems are suitable for our aim, because their development is explicitly based on design patterns and hence are good at evaluating design pattern mining approaches. Finally, 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.

According to what was discussed in the above sessions, we evaluate precision and recall of JHotDraw 6 and JUnit 3.7 comparing the results obtained by our tool with the ones obtained by an expert, that analyzed all the systems code and documentation. On the contrary for JHotDraw 7 and Avro we evaluate and compare obtained precision and recall using Tsantalis tool. The DPs that are considered in this investigation are the most common in literature: Factory, Prototype, Singleton, Adapter, State, Strategy, Composite, Decorator, Observer, Template, Command, Proxy, Visitor.
5 Discussion of Results

In this session we discuss the obtained results. First of all we evaluate the Precision and Recall of Design Pattern Finder in 2 OS systems. Successively, we evaluated Precision and Recall of Design Pattern Finder compared to Tsantalis. This comparison is obtained considering other 2 OS systems. Moreover, we report in this sessions some data about the performance of the proposed approach and finally we discussed the limits. For some of the considered DPs (i.e. Proxy), we didn’t find any occurrences. In the following tables we only show the DPs that were find.

5.1 Quantitative Analysis - Precision and Recall

A first validation of the approach was performed using the DPF tool to find the DPs contained in JHotDraw 6 and JUnit source code.

The Table 5.1, for each of the two analysed systems, reports: the name of the DPs searched in the code (first column); the number of each searched Pattern detected by the proposed approach (column D); the number of each searched Pattern the expert said to exist actually in the system (column Tp), the number of False Positive (column Fp), i.e. the number of DPs instances detected by the tool but rejected by the expert. The last two columns reports respectively precision (P) and recall (R) computed on the results as provided by the tool and validated by the expert.

As shown in Table 5.1, 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 number of false positives lower than patterns with a less constrained structure or with limited or absent behavioral properties.

The false negatives were related to patterns that were implemented differently from what assumed in the specification (that was based on the definitions given in literature, mainly [7, 10]). False negatives occurred for State, Singleton and Prototype DPs in JHotDraw 6, and for State and Observer DPs in JUnit. We found that the number of false negatives was dramatically reduced by adding new variants inheriting existing specifications and taking into account the structural differences that caused to miss them.

For some DPs we have an optimal recall but a precision less than 0.5; this was because some pattern specifications were particularly relaxed. As an example, it is the case of the Factory, in JHotDraw 6, where we considered a Factory even a class owning a single static method creating and returning an instance of another (abstract) class or interface. This explains why the Factory pattern has an optimal recall, but the precision is low.
Table 5.1: Precision and Recall for JHotDraw 6 and JUnit

<table>
<thead>
<tr>
<th>Design Pattern</th>
<th>JHotDraw 6</th>
<th>JUnit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D</td>
<td>TP</td>
</tr>
<tr>
<td>FactoryMethod</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Adapter</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Command</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Composite</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Observer</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Prototype</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Singleton</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>State</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>Visitor</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(less than 0.4): our pattern specification was particularly relaxed if compared to the Factory Method definition found in literature.

As results show, the proposed approach is able (depending on how specifications are written) 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 specification uses the invocation property requiring the Execute method, in the concrete subclass, to invoke a method of a class bound to a Command. The same happens for Composite/Decorator where the Decorator is required to specify a delegation towards the decorated object.

On the other two systems (JHotDraw 7 and Apache Avro 1.6) a comparison between results of our approach and those obtained using the similarity scoring approach of Tsantalis [26] has been performed. Tsantalis’ tool was chosen mainly because it adopts a similar technique (exploiting the same information, but used in a different way). Both the approaches are used to find DPs existing in Avro and JHotDraw7 code. For each tool, we evaluate Precision and Recall with respect to the gold standard calculated and previously discussed.

Table 5.2, for JHotDraw 7 and Apache Avro 1.6 systems reports the name of the detected DPs (first column) and the number of pattern considered as gold standard (GS). The remaining columns in the table, for each tool, reports the number of D (detected), TP (true positives), FP (false positives) and the computed values of Precision (P) and Recall (R).

The first consideration about the results is related to the presence of false positive and false negative. The ratio of false positive is less than 0.4% and 2% for respectively DPF and Tsantalis that is quite acceptable for both tools. However for some pattern,
and for both the approaches, the number of false positive is particularly higher than for other pattern. This happens, for DPF on JHotDraw 7, for Template Method patterns in which of 110 instances, 10 instances were not template methods. Inspecting those cases revealed a problem in the structure of the specification that was respected but was too weak and in some case, internal helper methods were considered as template methods. For the Observer pattern the results were similar, since the approach detected 105 observers instances (one for each concrete participant) but 9 of them were not Observers.

The Observer case is also interesting for what concerns the detection of patterns variants. DPF detected 96 true Observer instances on JHotDraw 7 (with 9 false positives) and 48 instances on Apache Avro 1.6 (without false positives). Our pattern specification repository actually was comprised of 3 variants for the Observer pattern. The first one exploits standard Java types (Observable class and Listener interface) while the other requires an abstract type for both subject and listener roles. The third one defines a multi-event observer in which the event context is created by a private helper method (fire(Event) methods) and passed to the corresponding callback on the listener (taking the context as parameter). The structure of this specification, reported in Figure 5.1, is quite different from the one proposed in literature. Inspecting the matched instances we found that, for the both JHotDraw 7 and Apache Avro 1.6 systems, the 96 and 48 instances respectively were all variants of the third type. This also explains why Tsantalis tool, that is based on the classic variant, was not able to find observers on these two systems.

5.2 Performance issues

We have measured the times of running the approach for each step of the detection process. Table 5.3 reports the execution times measured for each step of the detection
process and for each analyzed system. 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 and it resulted to be comparable to the other
structural approaches.

Experimentation performed for tuning the patterns specifications, showed that per-
formances can be considerably improved by identifying structural and behavioral con-
straints that are effective at identifying a well defined variant of a pattern. Hence the
approach is more effective when specifications are structured in a hierarchy and each
specification is dedicated to specific patterns variants.

5.3 Threats to Validity

This section discusses the main threats to the validity of our study. Construct validity
threats concern the relationship between theory and observation. There could be impre-
cisions/omissions in the measurements made in this paper for several reasons. One of the
most important limitation regards the generation of behavioral properties in presence of
late binding. In this case, as already stated, we have built a call graph using Rapid Type
Analysis (RTA) to reduce the set of possible callers. However the call graph still contains
a super-set of the actual calls. In the property extraction algorithm we decided to take
into account the sets of possible targets in order to perform the matches. In this way,
we surely don’t miss any possible binding but exposes the algorithm to the presence of
false positives (since the set of successful binding can be a super-set of the actual ones).
Further experimentation (with more strict policies) should be performed in order to as-
sess if the behavior of the algorithm improves with respect to this conservative choice.

Conclusion validity concerns the relationship between the treatment and the outcome.
As explained in Section 4, we performed a complete manual inspection only for two of
the four systems. For Apache Avro and JHotDraw7 systems, the gold standard was
computed comparing results obtained using Design Pattern Finder and Tsantalis’ tool.
This means that, for JHotDraw7 and Apache Avro 1.6 systems, we cannot exclude that
the computation of recall is imprecise (it could be higher than actual one since there
could exist pattern instances missed by both tools used in the study to compute the gold
standard). This, in future works, can be improved by performing a full analysis of the
searched source code base (or using available benchmarks). Threats to internal validity
concern factors that can influence our observations. In this case the identification of pat-
tern instances was based on the expert examination of internal/external documentation
and source code and hence could pose a threat to internal validity affecting the number
of false negatives. Threats to external validity concern the generalization of our findings.
Of course replication on further projects to confirm or contradicts the obtained results
is always desirable. Moreover we cannot claim that our approach produces the same
results on different (and larger) systems. Rather, we provide quantitative information
on the quality of the search for several real world systems and can affirm that precision
and recall have remained consistent and independent with respect to the system size. On the performance side there is a high dependency of the overall detection performance on the quality of pattern specifications. When specifications are badly written (that means few and overlapping constraints) the performance of the algorithm degrades rapidly.
<table>
<thead>
<tr>
<th>System</th>
<th>JHotDraw 7</th>
<th>Apache Avro 1.6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Design Pattern Finder</td>
<td>TSantalis</td>
</tr>
<tr>
<td>Tool</td>
<td>GS</td>
<td>D</td>
</tr>
<tr>
<td>Observer</td>
<td>96</td>
<td>105</td>
</tr>
<tr>
<td>Singleton</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Factory Method</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>Template Method</td>
<td>110</td>
<td>100</td>
</tr>
<tr>
<td>Adapter/Command</td>
<td>155</td>
<td>123</td>
</tr>
<tr>
<td>Decorator</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Prototype</td>
<td>46</td>
<td>34</td>
</tr>
<tr>
<td>State Strategy</td>
<td>194</td>
<td>168</td>
</tr>
<tr>
<td>Composite</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5.2: Precision and Recall for JHotDraw 7 and Apache Avro 1.6
<table>
<thead>
<tr>
<th>Step</th>
<th>Tool→</th>
<th>JUnit3</th>
<th>JHotDraw6</th>
<th>JhotDraw7</th>
<th>Avro1.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parsing &amp; AST extraction</td>
<td></td>
<td>157</td>
<td>1281</td>
<td>69974</td>
<td>255230</td>
</tr>
<tr>
<td>Meta-model generation</td>
<td></td>
<td>1199</td>
<td>7838</td>
<td>428147</td>
<td>1567785</td>
</tr>
<tr>
<td>Pattern repository detection</td>
<td></td>
<td>3502</td>
<td>11480</td>
<td>627090</td>
<td>2296271</td>
</tr>
<tr>
<td>Total Time</td>
<td></td>
<td>4858</td>
<td>20599</td>
<td>1125211</td>
<td>4120287</td>
</tr>
<tr>
<td>Average per pattern</td>
<td></td>
<td>540</td>
<td>2289</td>
<td>125023</td>
<td>457810</td>
</tr>
</tbody>
</table>

Table 5.3: Execution times of the design patterns detection process
6 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 Properties 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 in order to detect those components cooperating in a way that satisfies the model of a pattern. The approach also allows to identify DPs variants as modifications to pattern specifications already defined. The approach has been applied to two Java systems producing good results. Future work will be done to extend the catalog of the patterns to identify and to perform further experimentation on a wider set of open source systems.
Bibliography


