Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

On applying machine learning techniques for design pattern detection

Marco Zanoni*, Francesca Arcelli Fontana, Fabio Stella

Department of Informatics, University of Milano-Bicocca, Milano 20126, Italy

A R T I C L E I N F O

Article history:
Received 28 May 2014
Revised 19 January 2015
Accepted 20 January 2015
Available online 30 January 2015

Keywords:
Design pattern detection
Machine learning techniques

A B S T R A C T

The detection of design patterns is a useful activity giving support to the comprehension and maintenance of software systems. Many approaches and tools have been proposed in the literature providing different results. In this paper, we extend a previous work regarding the application of machine learning techniques for design pattern detection, by adding a more extensive experimentation and enhancements in the analysis method. Here we exploit a combination of graph matching and machine learning techniques, implemented in a tool we developed, called MARPLE-DPD. Our approach allows the application of machine learning techniques, leveraging a modeling of design patterns that is able to represent pattern instances composed of a variable number of classes. We describe the experimentations for the detection of five design patterns on 10 open source software systems, compare the performances obtained by different learning models with respect to a baseline, and discuss the encountered issues.

1. Introduction

Design pattern detection (DPD) is an active field of research, useful to support software maintenance and reverse engineering. In particular, it gives useful hints to understand some design decisions, which are helpful for the comprehension of the system and the following reengineering steps. Moreover, design pattern detection (DPD) helps also during the redocumentation phase of a system.

The detection techniques used in the literature span many areas, like graph theory, constraints satisfaction, fuzzy sets, machine learning and computation of similarity measures, but an optimal solution has not been found yet. Many causes concur to this situation, *e.g.*, design pattern definitions, programming languages. The definitions of design patterns, *e.g.*, the ones reported in Gamma et al. (1995), often focus on the pattern's *intent*, and less on its implementation, which is instead relevant from the reverse engineering point of view, because it is the input of the detection task. The different possible implementations of a design pattern are known as *variants* (Niere et al., 2002). Variants are mainly due to two factors: first, a single abstract solution can have different implementations, having exactly the same structure and behavior; second, as the definition of the pattern is informal, two different developers may have slightly different interpretations of the pattern definition, and they can produce different variants, having a different structure and, in theory, the same external behavior.

As the variants problems exists, also the interpretation and detection of design patterns can be subjective. In this context, a complete and closed specification of a pattern instance, used to perform an automatic detection, is not possible. We apply machine learning techniques to allow developers picking examples of what is a good or bad instance of a design pattern, and let a tool choose which features have to be considered for the detection. This approach allows to easily customize the detection, supporting also new patterns. With this approach, the set of correct and incorrect examples represents the informal specification (or detection rule) of a pattern. Hence, by changing the composition of example sets, it is possible to change the detection rule of patterns, *e.g.*, focusing the detection on different variants. The choices among correct and incorrect have to be performed by human experts. Example sets could be produced by a single development team, or could be available from the literature as a corpus, shared and agreed by many experts. Such a corpus is not available in the literature, at the best of our knowledge, so we created our set of examples for experimenting the learn-by-example approach described in this paper.

In our work, a module, called *Joiner*, extracts (possibly) all the pattern instances contained in a software system, while another module, called *Classifier*, classifies as correct or incorrect the instances detected by the Joiner, refining its results. The role of the Joiner is to find all the instances matching an exact rule. This rule is very general, and considers only the fundamental traits of the pattern structure. As a consequence, the matching tends to produce a large number of instances, achieving high recall, but resulting in low precision. Our rules for DPD are defined by exploiting some particular kind of micro-structures in the code, which can be

* Corresponding author. Tel.: +390264487848.
E-mail addresses: marco.zanoni@disco.unimib.it (M. Zanoni), arcelli@disco.unimib.it (F. Arcelli Fontana), stella@disco.unimib.it (F. Stella).

http://dx.doi.org/10.1016/j.jss.2015.01.037
0164-1212/© 2015 Elsevier Inc. All rights reserved.

unambiguously detected and capture relevant features of the pattern.

The separation of the approach in two phases (realized by the Joiner and the Classifier) has the benefit of submitting a limited number of candidates to the Classifier, but with a significant percentage of true instances. The choice of using a classification process after an exact matching is one of the main features characterizing our approach. Moreover, the process does not depend upon specific machine learning technologies, allowing the experimentation of different classifiers and clustering algorithms.

The main contributions of the whole DPD approach are:

- a modeling of design patterns able to represent instances composed of different numbers of elements, filling them in a structure taking into account the relationships among the different parts of a pattern;
- the formulation of the problem of DPD as a supervised classification task; this is possible due to the use of a pre-processing strategy capable to overcome the unstructured (from the data mining point of view) nature of design pattern instances;
- the construction of a large dataset of manually verified design pattern instances, which is useful and necessary for benchmark purposes.

We implemented the DPD approach based on machine learning techniques in our MARPLE project, with the name of MARPLE-DPD. In a previous paper (Arcelli Fontana and Zanoni, 2011), we introduced the general approach. Here, we extend and enhance the experimentation, by:

1. testing more and new patterns,
2. using more machine learning techniques,
3. testing algorithms on a larger dataset (publicly available), and
4. applying an automatic and systematic experimental method for the optimization of the algorithms' parameters.

Moreover, we enhance the detection process by introducing a custom clustering algorithm in the particular cases where the pattern structure is flat, without nesting levels. This enhancement has the goal to provide classifier algorithms with a more direct representation of pattern instances.

The approach is experimented for the detection of five design patterns (Singleton, Adapter, Composite, Decorator, Factory Method) on 10 open source software systems (having a total size of about 540 kLOC). The obtained performances are presented, using three different performance measures. The patterns to test were chosen by looking for the most frequently used, as described in the specialized literature.

In the experiments (see Section 6.2), we subdivided patterns in two groups. For Singleton and Adapter, we applied only classification models. For the second group, containing Composite, Decorator and Factory Method, we applied a cascade of clustering and classification models. We obtained good performances especially for Singleton, Adapter and Factory Method. From our evaluation, lower performances in the other patterns were due to the lack of available pattern instance examples. All the tested models performed better than the baseline model, except for Composite, where no statistically significant improvement was detected. In many cases support vector machines (SVMs), decision trees and random forests scored the best results. K-means was the only clustering model producing meaningful results.

The paper is organized as follows. In Section 2 we introduce some basic notions on DPD, relevant for the comprehension of our approach. In Section 3 we introduce the architecture and the main modules of MARPLE-DPD. In Section 4 we describe the modeling of design patterns and their detection process. In Section 5 we describe in detail the proposed classification process. In Section 6 we report the experiments we performed with different clustering and classification algorithms, outlining the performance differences among them. In Section 7 we outline the threats to validity of our approach. In Section 8 we introduce some related works on DPD, and in particular those based on approximated techniques. Finally, in Section 9 we give our conclusions and discuss the main future developments.

2. Background

In the following, we introduce some basic terminology on DPD and the source code features, *i.e.*, micro-structures, we use in our approach.

2.1. Code entities

Code entity is the name we give to any code construct that is uniquely identifiable by its name (and the name of its containers). In the object-oriented paradigm, code entities are classes (but also interfaces, enums, annotations, etc.), methods and attributes. The concept of code entity will recur because of the definition of micro-structures.

2.2. Micro-structures

To have an idea of what micro-structures (Arcelli Fontana et al., 2011a) are, it is possible to see them as facts about a pair of code entities: a *source* code entity and a *destination* code entity; the two code entities can also coincide. The definition of micro-structures allows to think about a software system as a graph, where code entities are associated with nodes, while micro-structures are associated with edges. Differently from design patterns, micro-structures are not ambiguous. Once a micro-structure has been specified in terms of the source code details used to implement it, it can be detected.

In our approach for DPD, we exploit different kinds of micro-structures: elemental design patterns (EDPs; Smith and Stotts, 2002), design patterns clues (Arcelli Fontana et al., 2011b, clues in the following) and micro patterns (Gil and Maman, 2005). Clues and elemental design patterns (EDPs) share the same detail level, as in general they can be detected by the analysis of single statements and elements of a class, like method invocations or field declarations. EDPs capture object-oriented best practices and are independent of any programming language; clues aim to identify basic structures peculiar to each design pattern. Micro patterns have been defined with the intent to capture recurrent patterns in the implementation of classes, *e.g.*, concerning the number of attributes or methods and their modifiers.

In spite of the differences between them, these micro-structures can be used for both the construction and the detection of design patterns. We provide, in Listing 1, an example of the "restricted creation" micro pattern.

An instance of this micro pattern can be found in classes without public constructors and with at least one static field of the same type of the class. Many classes following the Singleton design pattern satisfy these constraints, *e.g.*, `java.lang.Runtime`. Other examples of micro-structures are given in Section 4.1.4.

2.3. Design pattern detection

The definition of design patterns (Gamma et al., 1995) describes the *roles* of the elements composing them. Roles are the names of the tasks that each class (sometimes also methods) in a design pattern must accomplish. As there is a limited number of tasks to accomplish in a single design pattern, the number of roles in each pattern is *fixed*.

A *role mapping* is the assignment of a role to each class composing a pattern instance. In any design pattern instance, each role must be *mapped* to at least one class, *i.e.*, each class must have a role in the pattern. The extraction of role mappings allows the localization of design pattern instances in a software system, realizing the goal of DPD.

```
public class RestrictedCreationClass {
  static RestrictedCreationClass rcc;
  private RestrictedCreationClass() {
    ...
  }
}
```

Listing 1. Restricted creation micro pattern example.

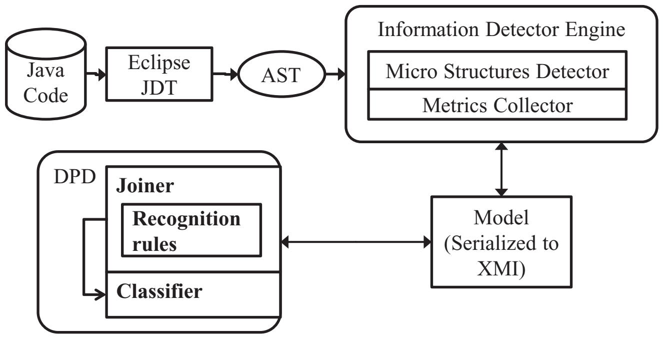
Java Code
Eclipse JDT
AST
Information Detector Engine
Micro Structures Detector
Metrics Collector
DPD
Joiner
Recognition rules
Classifier
Model (Serialized to XMI)

Figure 1. The architecture of MARPLE.

3. MARPLE-DPD

Metrics and ARchitecture Reconstruction PLugin for Eclipse (MARPLE)[1] is our tool implementing DPD from Java source code and other functionalities. The overall architecture of metrics and architecture reconstruction plugin for eclipse (MARPLE) is depicted in Fig. 1. DPD activities work on information extracted from the abstract syntax trees (ASTs) of the analyzed system. This information is represented by elements, or facts, which are called *micro-structures* (see Section 2), and by metrics that have been measured inside the system. Both these kinds of information are used to have an abstract and more consistent view of the system, instead of directly relying on the code or on the AST.

The architecture of MARPLE consists of three main modules that interact through a common model. Joiner and Classifier are the principal modules used for DPD and represent the MARPLE-DPD module.

- *Information detector engine*: Builds the model of the system and collects both micro-structures (through the micro-structures detector) and metrics (through the metrics collector), storing them in the model. This module implements the first level of abstraction provided by MARPLE.
- *Joiner*: Extracts all the potential design pattern candidates that satisfy a given definition, working on the micro-structures extracted by the information detector engine.
- *Classifier*: Makes inference on whether the groups of classes detected by the Joiner are (or not) realizations of design patterns. This module helps to detect false positives identified by the Joiner, and computes the similarity of the extracted patterns with known design pattern implementations, expressing it in the form of a confidence value.

The remainder of this section describes the Micro-structures detector (MSD), Joiner, and Classifier modules.

3.1. Micro-structures detector

In the micro-structures detector (MSD), micro-structures are extracted by a set of visitors parsing an AST representation of the source code, provided by the Java development tools (JDT)[2] Eclipse plugin; each visitor supports the detection of one micro-structure, and reports the found instances. The information is acquired through static analysis and is characterized by 100% rate of precision and recall. This value is due to the fact that these kinds of structures are meant to be *mechanically recognizable* (Gil and Maman, 2005), *i.e.*, there is always a 1-to-1 correspondence between a micro-structure and a piece (or a set of pieces) of code, and the algorithm that detects the micro-structure is defined in terms of the source code structure and properties. For the complete list of supported micro-structures, see Zanoni (2012).

Different micro-structures can be recognized in a software system using different techniques; the only constraint is that they have to refer to two (or also only one) code entities in the system. Code entities (see Section 2.1) are the central element of the representation model exploited in MARPLE. Micro-structures are currently detected by performing an exploration of the ASTs, but it is possible to exploit also (if needed) more complex analyses, *e.g.*, data flow analysis (Marlowe and Ryder, 1990), dynamic analysis (Ball, 1999).

3.2. Joiner

The Joiner analyzes the information coming from the MSD to extract groups of classes from the system, where each group is a pattern candidate. In a pattern candidate, a role is assigned to each class and the classes are organized according to a tree structure, which models the conceptual organization of the pattern. The same tree structure is used in our project for DPD tools benchmarking, and explained in Arcelli Fontana et al. (2012).

The Joiner represents the system as a graph, where the micro-structures extracted by the MSD are the edges and the nodes are the types. In this way, graph matching techniques can be applied for the extraction of pattern candidates.

In particular, the Joiner uses resource description framework (RDF) to represent the graph. The graph matching phase is implemented through SPARQL queries processed by the Jena ARQ[3] library.

3.3. Classifier

The Classifier evaluates the similarity of pattern instances to previously classified design patterns, deciding if the evaluated instances are correct or not. It also ranks them using a confidence value measuring the reliability of the decision. Instances can be inspected with different graphical views connected to the source code. After having inspected some instances, it is possible to evaluate them as *correct* or *incorrect*. Every time new instances are evaluated, it is possible to add them to the repository. At any time, the patterns in the repository can

[1] http://essere.disco.unimib.it/reverse/Marple.html.

[2] http://www.eclipse.org/jdt/.

[3] http://jena.apache.org/documentation/query/index.html.

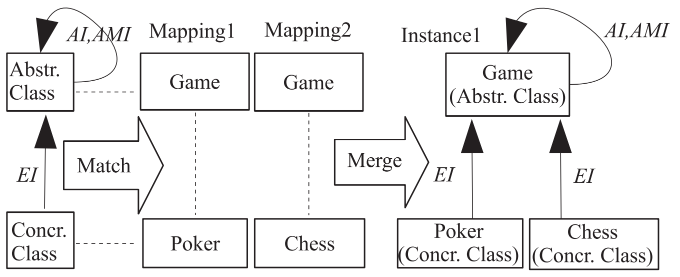

Figure 2. Joiner detection process example for the Template Method DP. Micro-structures: EI = extended inheritance, AI = abstract interface, AMI = abstract method invoked.

be used to train again the machine learning models, and apply the updated models in the detection process.

4. The detection process

The detection process of MARPLE-DPD is composed of different phases, supported by the different modules. In this section, we explain the sequence of operations we apply to detect design patterns.

4.1. Design pattern candidates extraction

The input of the detection process is the list of micro-structures detected on the system, in the form of a graph. The extraction of pattern instances is organized in two steps:

1. *Matching*: The application of graph matching produces a list of role mappings;
2. *Merging*: Role mappings are aggregated in pattern instances (candidates), following the specification of the pattern's tree-structure.

Fig. 2 informally depicts an example of the matching and merging process and results. The example shows the matching of the Template Method design pattern, defined on the left using two roles: *Abstract Class* and *Concrete Class*. In the example, two role mappings are extracted from a system during the matching step, and then merged in a single pattern instance during the merge step. In the following, we explain how the two steps work, and give more details about the reported example.

4.1.1. Matching step

The output of the matching step is a set of role mappings. The concept of role mapping is specifically instantiated in our case with the following properties: (i) each role mapping has a fixed length, equal to the number of roles declared in the design pattern definition, and (ii) each role is assigned to a class; the same class could be mapped to different roles. In other words, each role mapping is a function *rm* assigning a class to each role, as defined in Eq. (1), where *Roles* and *Classes* are the sets of the roles of the design pattern and the set of the classes of the system, respectively.

$$rm : Roles \rightarrow Classes \tag{1}$$

Matched (Eq. 2) is the set of all the *rm* mappings extracted for a pattern during the matching step.

$$Matched = \{rm_1, \ldots, rm_i, \ldots, rm_n\} \tag{2}$$

In Fig. 2, the match rule on the left defines that the class in the *Abstract Class* role must be abstract and must contain a method call to an abstract method of the same class, while the class in the *Concrete Class* role must inherit from the *Abstract Class*. These constraints are represented by the arrows connecting the two roles. The matched role mappings (Mapping1 and Mapping2) in the center of the figure assign the two roles respectively to classes (*Game*, *Poker*) and (*Game*, *Chess*).

The rules we define for the extraction of role mappings are kept very general, trying to cover all possible variants and to include all the existing pattern instances. The rules usually exploit a limited number of micro-structures; all the available micro-structures, instead, are used for the classification process. The matching of rules defined using these criteria tends to produce a large number of instances, but many of them are false positives, so precision is low and recall is high. The detection rules for all the design patterns defined in Gamma et al. (1995) are specified in Zanoni (2012).

4.1.2. Merging step

The structure used to represent pattern instances can be shortly defined using GEBNF (Bayley and Zhu, 2010):

$$\begin{aligned}
DpInstance &::= instance : LevelInstance, definition : \underline{DpDef}\\
DpDef &::= level : LevelDef, name : String\\
LevelDef &::= roleDefs : RoleDef^{+}, children : LevelDef^{*}\\
RoleDef &::= name : String\\
LevelInstance &::= levels : Level^{*}, roles : RoleAssociation^{+}\\
Level &::= instances : LevelInstance^{+}, definition : \underline{LevelDef}\\
RoleAssociation &::= role : \underline{RoleDef}, className : String
\end{aligned}$$

Design pattern instances are represented by *DpInstance* elements. A *DpInstance* (the starting symbol of the definition) refers to a *LevelInstance* with a *DpDef* definition. A *DpDef* defines the root of a tree of *LevelDef* nodes. Each node contains at least a *RoleDef*. A *LevelInstance* instantiates a particular set of *RoleAssociations* regarding the *RoleDef*s of its container *Level*. A *LevelInstance* contains other *Level*s, conforming to the respective *LevelDef*. *Level* elements simply act as containers of *LevelInstance* elements, and associate them to their respective *LevelDef*. *RoleAssociation* elements describe which role *RoleDef* is assigned to which class. The already defined *Roles* set contains *RoleDef* elements (see Eq. (1)). A role mapping can be seen as a set of *RoleAssociation* elements.

A relevant property of the proposed tree-wise modeling of design pattern instances is that the number of classes contained in an instance is allowed to be unknown at design time, while the number of the roles involved is known and given.

The merge procedure has the goal of filling a set of role mappings into the reported structure. It can be expressed with Algorithms 1 and 2, representing the *merge*() and *addLevel*() functions respectively. The *merge*() function is applied to all the *Matched* role mappings in a particular *dp* design pattern. The procedure returns the set of all the instances of *dp*. The algorithms work as follows. Given the set of *Matched* role mappings, and a root level definition, every role mapping is added as a child of the root level. For each role mapping *rm* and current *level*, a level instance *li* is retrieved, or created if not present. Level instances are identified in a level with the role associations regarding the roles set in the level definition. Classes assigned to roles in every level instance are the keys of the instance into its container level. When a level instance is created, it is initialized with its role associations (i.e., its key), and with an empty child level for each child level of its definition. When the level instance *li* is retrieved or created,

Algorithm 1 merge(m: *Matched*, dp: *DpDef*):

```
$l \leftarrow newLevel(dp.level)$
for all $rm \in m$ do
    $addLevel(rm, l)$ // add every mapping to the proper sublevel in $l$
end for
$merge \leftarrow l.instances$
```

```
SELECT  ?AbstractClass ?ConcreteClass
WHERE {
?AbstractClass
  AbstractMethodInvoked ?AbstractClass.
?AbstractClass
  AbstractInterface ?AbstractClass.
?ConcreteClass
  ExtendedInheritance ?AbstractClass.
}
```

Listing 2. Template Method matching rule.

Algorithm 2 addLevel(rm: RoleMapping, level: Level):

```
if $hasInstance(level, rm)$ then // get level instance if existing
    $li \leftarrow getInstance(level, rm)$
else // otherwise create a new one
    $li \leftarrow newLevelInstance(level)$
    for all $r \in level.definition.roles$ do // add its role mappings
        $li.roles \leftarrow li.roles \bigcup newRoleAssociation(r, rm(r))$
    end for
    for all $l \in level.definition.children$ do // and create its sublevels
        $li.levels \leftarrow \bigcup newLevel(l)$
    end for
end if
for all $sl \in li.levels$ do // add instances and mappings sublevels
    $addLevel(rm, sl)$
end for
```

all its children levels are recursively filled with the same procedure. The recursion stops when *li* has no children, i.e., when its container level definition has no children.

The algorithms uses some functions that we did not define. They are informally defined as follows:

- *newLevel*(l : *LevelDef*) creates a new *Level* instance referring to the *l LevelDef*;
- *newLevelInstance*(l : *Level*) creates a new *LevelInstance* instance and adds it to *Level l*;
- *newRoleAssociation*(r : *RoleDef*, c : *String*) creates a new role association for the specified role and class name;
- *hasInstance*(l : *Level*, *rm* : *RoleMapping*) tells if *l* already contains an instance having the same mappings (for the roles belonging to *level*) defined in *rm*;
- *getInstance*(l : *Level*, *rm* : *RoleMapping*) retrieves the *LevelInstance* in *l* having the same mappings (for the roles belonging to *l*) defined in *rm*.

In Fig. 2, the rule defines that the *Abstract Class* role is contained in the root *LevelDef* and the *Concrete Class* role in a child *LevelDef*. The containment is represented by the top-down placement of roles. The top role is in a parent level of the bottom role. The merge procedure puts together the two mappings in a single instance, because they have the same assignment of the *Abstract Class* role (which is defined in the root *LevelDef*). Two different *LevelInstances* are instead created, for the *Concrete Class* role. Adding a mapping having a different class in the *Abstract Class* role would create instead a new pattern instance.

4.1.3. Single-level and Multi-level patterns

Patterns are defined as trees of *levels*, each one containing at least a role. In this context, two relevant categories of patterns can be defined:

```
<pattern name="TemplateMethod">
  <role name="AbstractClass" />
  <sublevel>
    <role name="ConcreteClass" />
  </sublevel>
</pattern>
```

Listing 3. Template Method merging rule.

- *Single-level patterns*: Every pattern described by using only one level; this happens when there is a single role (*e.g.*, *Singleton*), or when different role mappings always produce different instances. In our definition, this happens, *e.g.*, for the *Adapter* design pattern, because it is considered an opportunistic[4] design choice; *Adapter* contains three roles in a single (root) level.
- *Multi-level patterns*: All the remaining patterns are multi-level, *i.e.*, they have at least one child level under the root one.

This categorization will impact the modeling process of the machine learning approach described in this paper.

4.1.4. Detection rule example: Template Method

In Listings 2 and 3 we report, respectively, the matching and merging rules for the *Template Method* design pattern, the one used as an example in Fig. 2. As we described in Section 3.2, matching rules are expressed using SPARQL.

The *Template Method* pattern is detected using a matching rule that specifies (1) classes declaring an abstract method and calling it, and (2) their subclasses. Here we briefly describe the micro-structures used in the rule:

- *Extended inheritance*: A design pattern clue, representing the transitive closure over the Inheritance EDP. The result is that every type (class or interface) is connected with every superclass or superinterface it is assignable to.
- *Abstract interface*: An EDP that is present when a class defines an abstract or empty method.
- *Abstract method invoked*: A design pattern clue, representing a method call made to an abstract method.

The merging rule defines that it is possible to have many *ConcreteClasses* for each *AbstractClass*. Template Method is an example of a multi-level pattern. The two roles are contained in two nested sublevels: *AbstractClass* is contained in the root level, represented by the `<pattern>` tag, and *ConcreteClass* is in a nested level, represented by the `<sublevel>` tag. The merging rules for the five patterns used in the experimentation described by this paper are reported in Appendix A.

[4] See the discussion of *Adapter* vs. *Bridge* in Gamma et al. (1995).

4.2. Classification process

MARPLE-DPD supports, through its classification facilities, an iterative and incremental design pattern discovery and evaluation methodology.

To accomplish its task, the classification process needs the following inputs:

- *Pattern definition*: The pattern definition given to the Joiner for its extraction task.
- *Micro-structures*: The set of the micro-structures collected on the analyzed projects.
- *Pattern repository*: A repository containing already classified pattern instances. The user can take advantage of an already filled repository, or contribute to it, or both.
- *Pattern candidates*: All the candidate design pattern instances identified by the Joiner module.

The Classifier exploits the information carried by the micro-structures available on the pattern instances returned by the Joiner. The Joiner exploits only a small subset of the available micro-structures, *i.e.*, the ones specified in the detection rule. The micro-structures already exploited by the Joiner are then discarded by the Classifier through feature selection.

The Classifier acts as a "filter" over the instances returned by the Joiner. Without the intermediate step implemented by the Joiner, we would need to submit every class (or every possible group of classes) of the system to the classification algorithm. Such an approach would be very computationally-demanding, and would require a lot of correct pattern instances to train the classifiers. The filtering approach we propose, instead, has the benefit of submitting a smaller number of candidates to the Classifier, and to create training sets containing a higher percentage of true instances. The overall process is defined through the following steps:

1. *From pattern instances to role mappings*
   - (1.a) pattern instances are represented using the mappings that contributed to their construction;
   - (1.b) each mapping is represented as a feature vector, as a result of the concatenation of the feature vectors representing the single classes in the mapping;
   - (1.c) mappings representing all extracted instances constitute a dataset; this dataset is called the *clustering dataset*;
2. *From role mappings to pattern instances*
   - (2.a) the clustering dataset is clustered into k clusters (with soft clustering);
   - (2.b) each pattern instance is represented as a feature vector, in which the features are created using the information coming from the clustering results;
   - (2.c) this last representation, called *classification dataset*, is taken as input by supervised classification algorithms; when classifiers are trained, a class label is applied to each feature vector. The class label tells if the pattern candidate is a correct instance or not.

In the following, we explain the process, and we describe how the datasets are built and processed to allow machine learning.

5. Design pattern instances modeling for classification

According to Witten et al. (2011), four basically different styles of learning appear in data mining applications, namely classification learning, association learning, clustering and numeric prediction. In this paper, we are concerned with classification learning and clustering. In classification learning, the learning scheme is presented with a set of classified examples from which it is expected to learn a way of classifying unseen examples, while in clustering groups of examples that belong together are sought. Classification learning is sometimes called supervised, because, in a sense, the scheme operates under supervision by being provided with the actual outcome for each of the training examples. This outcome is called the class of the example. The success of classification learning can be judged by trying out the concept description that is learned on an independent set of test data, for which the true classifications are known but not made available to the machine. The success rate on test data gives an objective measure of how well the concept has been learned. In many practical data mining applications, success is measured more subjectively in terms of how acceptable the learned description is to a human user. When there is no specified class, clustering is used to group items that seem to fall naturally together. The challenge is to find these clusters and assign the instances to them-and to be able to assign new instances to the clusters as well. The input to a machine learning scheme is a set of instances. These instances are the things that are to be classified or clustered. In the standard scenario, each instance is an individual, independent example of the concept to be learned. Instances are characterized by the values of a set of predetermined attributes. Each dataset is represented as a matrix of instances versus attributes, which in database terms is a single relation, or a flat file. In this format, each instance is represented by a feature vector, *i.e.*, a vector containing the values of the attributes (features) in the respective positions.

Table 1
Input format of a supervised classification algorithm having n features and a boolean class.

	f_1	…	f_i	…	f_{n-1}	f_n	Class
I1		…		…			(T) *correct*
I2		…		…			(F) *incorrect*
I3		…		…			(F) *incorrect*

Table 1 shows an example of a dataset having n features, from f_1 to f_n plus a boolean {*correct*, *incorrect*} class attribute. The dataset represents three design pattern instances, one correct and two incorrect. A look to the dataset format makes the modeling problem clear. We need to represent a pattern instance as a *feature vector*, but a design pattern instance is a *group* of classes, of *unknown* size, and organized in a tree structure (see Section 4.1). So our problem is to find a finite set of features that can be extracted from the source code, but able to describe the pattern instance as a whole.

It would be logical to use micro-structures as features into our dataset representation, because they allow the description of different aspects of the analyzed system, exposing different properties independently and with the same syntax. In fact, we can easily represent micro-structures as boolean features about pairs of classes. Our problem, instead, is the classification of pattern instances, which are groups of classes that need to be expressed as feature vectors. Therefore, we need a way to transfer the information provided by micro-structures from the classes to the pattern instances.

In the remainder of this section, we will exploit the roles defined in design patterns to build the representation of pattern instances, following the steps introduced in Section 4.2. We also provide in Fig. 3 an example graphical representation of the steps we are going to detail, reporting the step numbers along with the labels of each transformation. For simplicity, the example uses a virtual[5] pattern with two roles R_1, R_2; the instances used in the training phase are the ones shown in Table 1. We also represent only three micro-structures, from M_1 to M_3, while in MARPLE we support more than 70 different micro-structures.

[5] Using a real pattern definition at this stage would unnecessarily complicate the figure.

Table 2
Clustering dataset example.

Map#	$R_1M_1R_1$	$R_1M_1R_2$	$R_1M_2R_1$	$R_1M_2R_2$	$R_1M_3R_1$	$R_1M_3R_2$	$R_2M_1R_1$	$R_2M_1R_2$	$R_2M_2R_1$	$R_2M_2R_2$	$R_2M_3R_1$	$R_2M_3R_2$
1	1	1	1	0	0	1	1	0	1	1	0	1
2	0	1	1	0	0	0	1	0	1	0	1	1

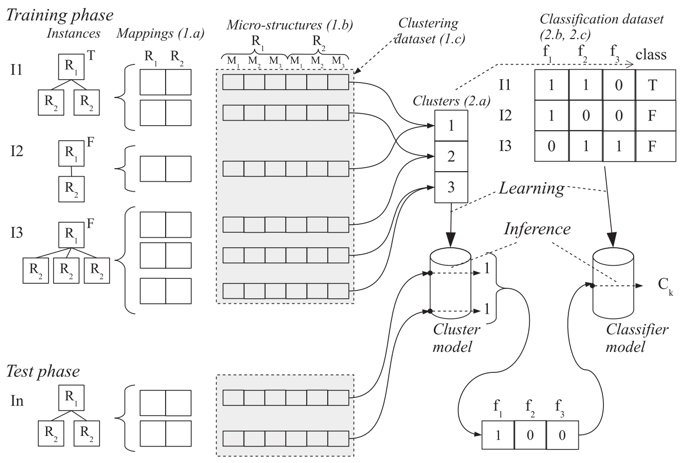

Figure 3. Complete classification process (simplified: the complete representation of the clustering dataset is shown in Table 2 for space reasons).

In the training phase shown in the example, each of the three pattern instances are decomposed in their role mappings, which are then represented as feature vectors. Feature vectors, composed of the micro-structures existing in the mappings, are used to train a cluster model. The three resulting clusters are used as features (f_1, f_2, f_3) to build a new feature vector, one for each pattern instance. This last representation is used to train a classifier model.

In the test phase, an unclassified new pattern instance (In) is decomposed in two feature vectors, that are clustered using the previously built cluster model, creating a representation suitable for the classifier model, which assigns class C_k to In.

5.1. From pattern instances to role mappings

Given the extraction process performed by the Joiner, it is possible to track the set of role mappings that concur to the composition of a single pattern instance. Therefore, we can represent a pattern instance by using the list of the role mappings used to create it (Step 1.a). Composing this kind of representation with the information coming from the micro-structures representation of classes (Step 1.b), a new representation of mappings as dataset rows is achieved (the *clustering dataset*), like the one shown in Table 2 (Step 1.c). In the example, the two mappings of instance I1 (see Fig. 3) are represented by 12 features, created combining two roles $R = \{R_1, R_2\}$ with three micro-structures $M = \{M_1, M_2, M_3\}$.

The feature set is defined as $R \times M \times R$. In the example, in fact, there are $2 \times 3 \times 2 = 12$ features. Every feature has a boolean value (in the examples we use 0=false and 1=true), and is defined as $R_iM_jR_k$. $R_iM_jR_k$ is true when, in a mapping, the class in role R_i is the source of a micro-structure M_j, having as destination the class in the R_k role of the same mapping.

Micro-structures are *binary* relationships (or facts) among classes or other code entities, but they can be unary when the source and destination of the micro-structure are the same; some micro-structures are unary by definition, *i.e.*, all the ones describing a simple property of a class. Unary micro-structures allow the reduction of the features actually used in this representation. In fact, knowing that M_j can exist only on a single code entity, we can remove every feature $R_iM_jR_k$ where $i \neq k$, obtaining a more compact representation. An example of micro-structure of this kind is the restricted creation micro pattern shown in Section 2.2.

In the clustering dataset the information about pattern instances is not exploited yet: the rows represent the mappings, not the instances. There is no link to the instance the mappings belong to, and no class label; these pieces of information are kept out of the clustering dataset, and exploited during the final classification phase.

The clustering dataset cannot be used for direct supervised classification of pattern instances, but it is suitable as input to machine learning algorithms.

5.2. From role mappings to pattern instances

Exploiting the representation given by the clustering dataset, the next step (2.a) groups the mappings in k clusters through a clustering algorithm, and represents each pattern instance using the set of clusters its mappings belong to. For example, in Fig. 3 the mappings are grouped in three clusters. The choice of the clustering algorithm (and k) is left out of the process, to allow the usage of different algorithms. Soft clustering is admitted, to achieve flexibility in the overall process.

In the next step (2.b), each pattern instance is represented as a boolean vector in a classification dataset, consisting of k features, *i.e.*, one for each cluster. The ith cell of each vector tells if the instance contains a mapping that was clustered in cluster i. The result is the *classification dataset*, where, if needed, the class label is added to each vector. Classifiers can be applied to data in this form, for training or for evaluating new instances (Step 2.c). For example, in Fig. 3 the two mappings of instance I1 are clustered in clusters 1 and 2, respectively. Therefore, I1 has features f_1 and f_2 set to 1 in the classification dataset, while feature f_3 is set to 0. The class of I1 (T) is then reported in the dataset, and the representation is complete.

The whole process is applied exactly in the same way for training and test instances. Obviously, test instances have no class attribute in the final dataset, and they are clustered and classified with models that have already been trained.

The process handles the problem of the unknown size of the pattern instance, by selecting a number of clusters, and therefore by limiting to a fixed number the features representing a single pattern instance. The classification dataset is an encoding of the role mappings, which uses information directly retrieved from the system in the form of micro-structures. The dataset combines information about the structure and the roles of the classes contained into each pattern instance, but limits the description of instances to a fixed size.

6. Experimenting MARPLE-DPD

Numerical experiments have been performed on the datasets representing the following patterns: *Singleton*, *Adapter*, *Composite*, *Decorator*, *Factory Method*. The way patterns are modeled in MARPLE-DPD, and an overview of the micro-structures exploited by the Joiner for each pattern, are described in Appendix A.

The patterns to test were chosen as the most frequently used, as described in the specialized literature. The choice was made by analyzing the results coming from the pattern-like micro-architecture repository (P-MARt) dataset proposed by Guéhéneuc (2007), the DPD tool from Tsantalis et al. (2006) and DPD results gathered by Rasool and Mäder (2011) and published by Software Engineering Research Center (2012). The experiments were performed using MARPLE version 2013-02.[6] This section explains some of the performed experiments and reports the obtained results.

6.1. Experiments

Numerical experiments are devoted to investigate the accuracy achieved by different machine learning models. Their goals are: (i) testing the superiority of machine learning models with respect to a baseline model, (ii) selection of the most effective machine learning models and (iii) comparison of alternatives for data pre-processing, *i.e.*, the selection of the clustering algorithm.

The experiments on the five design patterns were conducted applying a set of clustering and classification algorithms to a set of projects, and making an optimization of the parameters for each one of the algorithms, with respect to three different performance measures. For each considered performance measure, a different optimization of the parameters was performed. To produce the values of the performance measures, we used 10-fold cross validation. We relied on the cross validation procedure available in Weka (Hall et al., 2009). The procedure automatically randomizes the dataset before creating folds.

For each pattern, a set of instances were extracted from the projects by using the Joiner module, and then manually classified, producing the dataset we used as oracle during the cross-validation. The cross validation procedure considers the dataset as a single data source, so instances coming from different projects are grouped in a single set. The randomization applied before the construction of the folds has the goal of avoiding the possible bias introduced by the characteristics of the single projects.

6.1.1. Pattern evaluation criteria

The manual evaluation of the design pattern instances is a task with a high degree of subjectivity. Design pattern definitions do not allow to formally decide if a candidate represents a correct instance of a particular pattern. Moreover, some candidates are difficult to classify, for their complexity or because they allow for different interpretations of the design pattern definitions.

During our manual evaluations, we have been keeping track of some criteria we applied, to avoid producing conflicting evaluations in similar situations. A general rule we followed is: consider as *correct* the pattern instances containing all the correct key/important roles, even if they contain other noise. This approach is targeted to the collection of all design patterns in a system: the user can decide which part of the pattern instance is worth investigating, if the important roles are correctly assigned.

In the following, we list some of the other considerations we recorded during the manual evaluation, while a wider discussion is reported in Zanoni (2012).

- *Singleton*: An empty public constructor makes Singleton impossible;
- *Adapter (class)*: At least one method inherited from Target should call a method inherited from Adaptee;
- *Adapter (object)*: The field referencing the Adaptee may not be changed before its usage by the Target method overridden implementation: in other words the state of the Adaptee field should be immutable.

6.1.2. Dataset for experiments

A set of 10 projects were used for the gathering of design pattern instances, composed of: one project containing example pattern instances gathered on the web and nine projects contained in P-MARt. The summary of the experimented systems and some metrics describing them are shown in Table 3.

Table 3
Projects for the experimentations.

Project	CUs	Pack	Types	Meth	Attr	TLOC
DPExample	1060	235	1749	4710	1786	32, 313
QuickUML 2001	156	11	230	1082	421	9233
Lexi v0.1.1 alpha	24	6	100	677	229	7101
JRefactory v2.6.24	569	49	578	4883	902	79, 732
Netbeans v1.0.x	2444	184	6278	28,568	7611	317,542
JUnit v3.7	78	10	104	648	138	4956
JHotDraw v5.1	155	11	174	1316	331	8876
MapperXML v1.9.7	217	25	257	2120	691	14,928
Nutch v0.4	165	19	335	1854	1309	23,579
PMD v1.8	446	35	519	3665	1463	41,554

CUs: Number of compilation units, TLOC: Tot number of lines of code.
Pack: Number of packages.

Table 4
Summary of detected pattern instances.

Pattern	Candidates	Evaluated	Correct	Incorrect
Singleton	154	154	58	96
Adapter	6733	1221	618	603
Composite	128	128	30	98
Decorator	250	247	93	154
Factory method	2875	1044	562	482

Table 4 reports the amount of instances found by the Joiner (Candidates) and evaluated for each pattern. Evaluated instances were found either correct or incorrect. The number of evaluated instances for the *Adapter* and *Factory Method* patterns is lower than the number of candidates.

We decided to limit the number of evaluated instances around one thousand, which we considered a good threshold, allowing to create a dataset able to give significant results, while limiting the effort of the manual evaluation.

The evaluated pattern instances are available through the design pattern detection tools benchmark platform (DPB; Arcelli Fontana et al., 2012), available at http://essere.disco.unimib.it/DPB. In the platform, evaluations are classified under the tool named “MARPLE-DPD 0.0.20120718.dpd”. This is, to our knowledge, the largest dataset of manually evaluated pattern instances available, considering the five supported patterns. As a support to this statement, we supply in Table 5 a summary of design pattern instance datasets referenced in the literature. Some of these datasets are not publicly available, and some are no more available. Moreover, an advantage of exposing the dataset in design pattern detection tools benchmark platform (DPB) is in having structured and browseable data representation.

6.1.3. Algorithm selection and parameter optimization

The following machine learning models have been used for numerical experiments: ZeroR, OneR, Naïve Bayes, JRip, Random Forest, C4.5, SVMs (with different kernel functions), SimpleKMeans, CLOPE. The implementation of the algorithms is the one available in Weka. Refer to the tool’s website[7] for the full documentation and references.

The motivations for this selection are due to opportunity and to the fact that selected models represent the main approaches to supervised classification, namely probabilistic, separation and heuristic approaches. Furthermore, the specialized literature on DPD discusses some of these machine learning algorithms for the considered task, *e.g.*, Ferenc et al. (2005), Guéhéneuc et al. (2004), and Dong et al. (2008). ZeroR and OneR are included with the purpose of using them as baselines. ZeroR always chooses the class having the maximum a priori probability, so its accuracy is a measure of the balancing of the

[6] http://essere.disco.unimib.it/reverse/Marple.html.

[7] http://www.cs.waikato.ac.nz/ml/weka/.

Table 5
Evaluated dataset available in the literature.

Dataset	Patterns	Projects	Instances	Evaluation
P-MARt	22	9	139	Format: XML. Manually detected and validated
Pattern inference and recovery tool (PINOT)	17	9	4580	Format: plain text. Automatically detected, not validated
SERC	18	7	328	Format: PDF. Automatically detected, validated, considering other SSA and PMART published outputs as references.
DPRE	–	–	–	web page unavailable as of 2014-03-21
DPJF	7	6	108	Format: SWI-Prolog files (private communication). Manually and automatically detected, validated.
SSA	9	3	167	Format: web page report. Automatically detected, validation not explicitely reported.
Pettersson	–	–	–	web page unavailable as of 2014-03-21
MARPLE-DPD	5	9	2794	Format: web application, export in XML. Automatically detected, validated.

P-MARt (Guéhéneuc, 2007); pattern inference and recovery tool (PINOT) (Shi and Olsson, 2006); SERC (Rasool and Mäder, 2011); DPRE (De Lucia et al., 2010); DPJF (Binun and Kniesel, 2012); SSA (Tsantalis et al., 2006); Pettersson (Pettersson et al., 2010).

dataset. In our specific case, its accuracy is also the accuracy of the Joiner, which creates the input dataset. OneR selects the most representative single feature for classification: its performances give an idea of the difficulty of the classification task, and the gain obtained by other classifiers.

The evaluations were then exploited to search the best classification setup, *i.e.*, the assignment of parameters for each learning algorithm producing the best performance values.

The search for the best parameters of a machine learning algorithm can be a long, tedious, and error-prone task (when results are recorded manually). Each algorithm has parameters with different domains (*e.g.*, continuous, discrete, nominal) and ranges. The number of clusters, *e.g.*, is one of the typical parameters of this class of algorithms, and often one of the most important ones. The entire set of parameters of a single algorithm is a potentially huge space to explore. One traditional way of exploring the space is to perform a grid search,[8] which means fully exploring the search space with coarse granularity, and then focusing on the best region with finer granularity.

A faster process was needed in our case, and an approach exploiting genetic algorithms was taken to reach the objective. In particular, we exploited the JGAP[9] library, which allows the abstraction from the algorithm details, providing a simple way of extending the framework by adding a new fitness function. The same kind of approach was described and tested in the literature, *e.g.*, by Samadzadegan et al. (2010).

We defined a fitness function, performing 10-fold cross validation, and returning the value of one of the considered performance measures, *i.e.*, accuracy, F1-measure and area under ROC (AUC; Witten et al., 2011). Accuracy is the ratio of correctly classified instances, in both the positive and negative classes. F1-measure (also known as F1-Score or F-Measure) is equal to $2 \cdot precision \cdot recall / (precision + recall)$; in our experiments, we consider always the F1-measure value for the positive class. area under ROC (AUC) is the area under the receiver operating characteristic (ROC) curve; it grows near to value 1 when the discrimination performs better, while bad classification brings values near to 0.5.

The three performance measures have been optimized separately on every considered learning algorithm, producing a different parameter configuration for each of algorithm-measure combination. The performance measures we rely on are the ones, available in the Weka framework, which provide a single score value, as opposed, *e.g.*, to using precision and recall; in fact, only single scores can be used as values of a fitness function. The usage of three different performance measures allows us to evaluate the models from different perspectives.

6.1.4. Clusterer choice for single-level patterns

One of the decisions needed to instantiate the process over a particular design pattern is the choice of the clustering algorithm. For single-level patterns, we apply a simple custom clustering algorithm that exploits the properties of this class of patterns.

We cluster role mappings with the aim of representing any pattern instance as a feature vector of fixed size. Since pattern instances can be composed of many role mappings, we cluster role mappings in k clusters, to obtain k features in the vector, to be used for both learning and classification. However, in a single-level design pattern all the roles are defined in the root of the tree representing the pattern. This property leads to design pattern instances composed of a single role mapping. As a consequence, every single-level pattern instance would be represented using a feature vector always representing a single mapping, i.e., having only one *true* feature.

Instead, we choose to exploit this property. When a single-level design pattern is analyzed, the clusterer produces a classification dataset having the same number of features as the clustering dataset, using this simple criterion: cluster i contains a mapping if the value of the ith feature of the mapping is *true*.

In this work, we applied this clustering choice to the *Singleton* and *Adapter* design patterns, the only two single-level patterns among the five we experimented.

6.2. Results

For each design pattern, the classifiers' setups achieving the best performance are reported and discussed, together with their performance values. We report the best accuracy, F1-measure, and AUC values obtained by each considered clustering and classification algorithm combination. Accuracy values are compared by means of hypothesis testing, to decide if some algorithms are significantly better than the baseline, or than the others. We applied the test described in Witten et al. (2011, pp. 157–159), with nine degrees of freedom, since we applied 10-fold cross validation. The experiments with the Classifier module are made on the output of the Joiner: Every direct or indirect performance value (including *recall*) is calculated on that particular dataset. In other words, the Classifier module performance is estimated under the hypothesis that the Joiner succeeded in extracting every pattern instance contained in the analyzed systems, as explained in Section 4.2.

In the results, some classifiers (J48, C-SVC, μ-SVC) are reported in different variants; the reason of this choice is that each variant represents a major option of the algorithm having different settings, and leading to major differences in the algorithms being applied, *e.g.*, the pruning strategy in J48 and the SVM type and kernel in LibSVM.

6.2.1. Single-level patterns

Results of numerical experiments are summarized in Table 6. *Singleton* and *Adapter* are single-level patterns, so only classifier algorithms were experimented.

[8] as suggests, *e.g.*, http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf.
[9] http://jgap.sourceforge.net/.

Table 6
Best performances on single-level design patterns.

Classifier	Singleton			Adapter		
	Acc.	F_1	AUC	Acc.	F_1	AUC
ZeroR	0.61	0.00	0.48	0.53	0.00	0.50
OneR	0.87	0.83	0.86	0.68	0.66	0.68
NaiveBayes	0.81	0.73	0.89	0.70	0.70	0.78
JRip	0.88	0.85	0.88	0.81	0.77	0.82
RandomForest	**0.93**	0.90	**0.97**	0.85	0.84	0.92
J48 Unpruned	0.87	0.82	0.91	0.80	0.79	0.86
J48 Reduced Error Pruning	0.88	**0.91**	0.85	0.79	0.79	0.84
J48 Pruned	0.88	0.84	0.91	0.80	0.79	0.86
SMO RBF	0.90	0.86	0.94	0.85	0.84	0.92
C-SVC Linear	0.83	0.77	0.87	0.80	0.79	0.85
C-SVC Polynomial	0.91	0.88	0.94	0.84	0.82	0.89
C-SVC RBF	0.92	0.89	0.95	**0.86**	**0.85**	0.92
C-SVC Sigmoid	0.86	0.88	0.94	0.79	0.69	0.84
ν-SVC Linear	0.91	0.88	0.93	0.80	0.79	0.85
ν-SVC Polynomial	0.90	0.87	0.94	0.84	0.82	0.91
ν-SVC RBF	0.91	0.89	0.95	**0.86**	0.84	**0.93**
ν-SVC Sigmoid	0.90	0.79	0.92	0.75	0.64	0.82

Acc: Accuracy, F_1: F1-measure, AUC: area under ROC.

A first remark is related to the effectiveness of the machine learning approach to DPD. Accuracy values achieved by all machine learning models are significantly better than the accuracy value achieved by the baseline classifier (ZeroR), according to hypothesis testing applied to mean accuracies difference between ZeroR and each classifier with 95% confidence (data not reported here).

Hypothesis testing on mean accuracies difference at the 95% of confidence applied to the *Singleton* dataset allows the conclusion that most models have similar performances, with the exception of Naïve Bayes and C-SVC linear. These two models have significantly lower performances than the others. The analysis of the *Adapter* dataset allows to conclude that the Random Forest, SMO RBF, C-SVC polynomial, C-SVC RBF, ν-SVC polynomial and the ν-SVC RBF achieve comparable accuracy values, which are greater than those achieved by the remaining machine learning models.

The results achieved on the *Adapter* and *Singleton* design patterns suggest that the custom clustering algorithm applied to single-level design patterns is reasonable. In fact, different algorithms are able to reach good performances on the two datasets.

6.2.2. Multi-level patterns

The detection of multi-level patterns needs clustering algorithms to set the features used for learning to a fixed number. A smaller number of classifier models have been used in this experimentation, exploiting what emerged from the results of single-level patterns. In particular, we applied SimpleKMeans, which is an implementation of the K-means algorithm, and CLOPE (Yang et al., 2002), a clustering algorithm for transactional data; a peculiarity of CLOPE is being designed for nominal attributes datasets.

Table 7 contains the best results for the remaining three patterns. In the table, every number refers to the best parameter setup found for the respective datasets, algorithm and performance measures. We recall that the number of clusters is one of the parameters (only in SimpleKMeans), and is not reported for space reasons.[10]

Hypothesis testing on mean accuracies difference tells us that the machine learning models were effective for detecting the *Decorator* and *Factory Method* patterns, while this statement does not hold for the *Composite* dataset. Moreover, the best results achieved for the *Composite* pattern are by far lower than the ones for the other design patterns. The maximum performance values are scored by SVMs. One of the motivations of the low performances on this pattern could be the insufficient information contained in the dataset.

A second remark is related to pre-processing the dataset with clustering. Indeed, the K-means algorithm allows the achievement of better accuracy values for the *Decorator* and *Factory Method* patterns while it does not bring any advantage with respect to the CLOPE algorithm in the case when the *Composite* pattern is analyzed.

We applied hypothesis testing on mean accuracies superiority (one-sided hypothesis test) with 5% of significance. On the *Decorator* pattern, in the case the K-means pre-processing algorithm is used, OneR achieves a smaller accuracy value than the other machine learning models. For the *Factory Method* pattern, we conclude that the Naïve Bayes, Random Forest, J48 pruned and un-pruned, C-SVC RBF and the ν-SVC RBF achieve comparable accuracy values, which are greater than those achieved by the other machine learning models. The comments related to the *Decorator* and *Factory Method* patterns concern the use of the K-means pre-processing algorithm while, as pointed out, the use of CLOPE does not bring to interesting results.

The number of instances found in the target systems for the *Factory Method* design pattern is huge compared with the *Composite* and *Decorator* patterns. Moreover, in the dataset we have 52% of positive instances; this fact explains why ZeroR has F1-measure > 0: it always chooses the positive class, so it has low precision, but bad recall. One of the causes of the size of the dataset is the large diffusion of this pattern, which simply prescribes an indirection in the responsibility of the creation of a new object, and another one is that the rule groups *Factory Method* instances using the *Creator*, *ConcreteCreator* and *Product* roles in the root level (see Appendix A). The rationale of this grouping choice, reported in Zanoni (2012), is based on the definition and intent of the pattern itself. The sum of these causes led to 1044 pattern instances evaluated out of 2875 found ones.

6.3. Discussion on the obtained performances

The performances obtained by the different classifiers on the five datasets appear to be influenced by three main factors:

- Usage of clustering
- The kind of applied clustering algorithm
- The size of the dataset

Single-level patterns reached better average performances than multi-level ones. No real clustering algorithm was needed for these patterns, so the classification algorithms could directly work on the representation of the system provided by the micro-structures.

In multi-level patterns two apparent factors are influencing the obtained performances. The first is the kind of clustering algorithm: SimpleKMeans appears to perform better on this kind of data than CLOPE, especially when the dataset size grows. The other is the size of the dataset: *Composite*, *Decorator* and *Factory Method* have increasing dataset sizes, and increasing performances. One hypothesis is that on large datasets the clustering solutions provide a better description of the input, thus the classification can work on data better representing the underlying code. Another hypothesis is that the behavior simply depends on the pattern, and not on external causes. These hypotheses could be verified by computing the learning curves of all the algorithms, and is planned in future work.

The recall of the detection process is hard to establish. There are no shared and agreed baselines or benchmarks telling which patterns instances exist in different systems. We performed an evaluation of MARPLE-DPD on JHotDraw v5.1, and compared the results of the MARPLE Joiner and Classifier modules with the ones obtained by other tools: P-MARt (Guéhéneuc, 2007), SSA[11] (Tsantalis et al., 2006), web of patterns (WoP)[11] (Dietrich and Elgar, 2007), detection

[10] Parameters have been optimized for each algorithm-pattern-measure combination, so we would need to report the value of k (number of clusters) for each reported performance value.

[11] Using results loaded into http://essere.disco.unimib.it/DPB/.

Table 7
Best performance results for the multi-level patterns.

Classifier	Composite			Decorator			Factory Method		
	Acc.	F_1	AUC	Acc.	F_1	AUC	Acc.	F_1	AUC
Clusterer: SimpleKMeans									
ZeroR	0.75	0.00	0.39	0.58	0.00	0.49	0.52	0.69	0.50
OneR	0.75	0.11	0.50	0.70	0.56	0.65	0.70	0.73	0.70
NaiveBayes	0.77	0.38	0.63	0.77	0.74	0.76	0.81	0.82	**0.87**
JRip	0.75	0.35	0.48	0.80	0.76	0.76	0.76	0.79	0.77
RandomForest	0.75	0.45	0.61	**0.82**	**0.77**	**0.82**	**0.82**	**0.83**	**0.87**
J48 Unpruned	0.75	0.27	0.59	0.77	0.75	0.74	0.80	0.81	0.86
J48 Reduced Error Pruning	0.77	0.25	0.51	0.77	0.73	0.77	0.78	0.81	0.85
J48 Pruned	0.75	0.00	0.51	0.80	0.76	0.75	0.79	0.81	0.85
C-SVC RBF	0.79	0.36	**0.88**	0.80	0.75	**0.82**	**0.82**	**0.83**	0.84
ν-SVC RBF	**0.81**	0.55	0.67	0.80	0.76	0.81	0.80	0.82	**0.87**
Clusterer: CLOPE									
ZeroR	0.75	0.00	0.39	0.58	0.00	0.49	0.52	0.69	0.50
OneR	0.75	0.00	0.50	0.66	0.59	0.64	0.54	0.69	0.53
NaiveBayes	0.75	0.12	0.52	0.70	0.67	0.73	0.55	0.69	0.55
JRip	**0.81**	0.42	0.60	0.71	0.65	0.68	0.54	0.69	0.52
RandomForest	**0.81**	0.38	0.65	0.73	0.72	0.74	0.55	0.69	0.57
J48 Unpruned	**0.81**	0.42	0.53	0.73	0.66	0.74	0.55	0.69	0.54
J48 Reduced Error Pruning	**0.81**	0.25	0.56	0.72	0.68	0.74	0.55	0.69	0.55
J48 Pruned	0.75	0.42	0.58	0.73	0.65	0.73	0.55	0.69	0.54
C-SVC RBF	**0.81**	0.40	0.61	0.72	0.66	0.72	0.56	0.69	0.54
ν-SVC RBF	0.77	**0.56**	0.72	0.74	0.70	0.76	0.56	0.69	0.58

Legend: Acc: Accuracy, F_1: F1-measure, AUC: area under ROC.

of patterns by joining forces (DPJF; Binun and Kniesel, 2012, results exported by the authors of the tool), and our manual evaluation.

Considering the union of all the reported results, and our manual evaluation of the correctness of every reported instance, we found that the Joiner missed some correct instances only for the Adapter pattern. Nine out of the 71 correct instances were not extracted from the system; the resulting recall is 87% for Adapter, and 100% for the others. The errors are due to the exclusion of interfaces as Adaptees; their inclusion would have included a large amount of system interfaces (e.g., `Serializable`) as Adaptees, with a huge increase of false positives.

7. Threats to validity

In the following, we outline the relevant threats to the validity of our work.

7.1. Threats to external validity

A first threat to the generalization of our results is related to the construction of the training sets for the supervised classification algorithms. The training sets are based on a manual design pattern labeling task. The labeling was performed while paying attention to give objective evaluations, but it can contain some degree of subjectivity. The approach is designed to perform detections guided by the composition of the training set. Hence, the outcomes of our experiments are clearly influenced by the way datasets were created. For this reason, the datasets are publicly available on the DPB[12] platform, to allow understanding the reasons of the obtained performance measures.

Another consideration is that the Classifier performances are estimated under the assumption that the Joiner has 100% recall. To ensure this assumption, we manually tested our detection rules, ensuring they were able to recall every pattern instance known by prior knowledge, *i.e.*, found through a manual inspection of the system and the application of other DPD tools. In this way, we addressed the problem related to the unknown number of pattern instances contained in the analyzed projects.

[12] http://essere.disco.unimib.it/DPB.

The last threat to external validity is related to the size of the training datasets. More systems and design patterns can be analyzed to verify the experimental results. The reported experiment uses a well-known set of systems of non-trivial size.

7.2. Threats to internal validity

In the current implementation, MARPLE does not handle the contents of libraries, with the exception of inheritance. This behavior leads to a partial representation of some classes or subsystems, lowering the precision of the description of the system fed into the classifier. As the source code of libraries is not guaranteed to be available, the issue can be handled in the future, by letting the user decide if to retrieve pattern instances partially belonging to a library or to concentrate on the ones strictly contained in the source code.

8. Related work

According to the approach described in this paper, we now briefly describe only the tools providing some kind of approximated recognition, as in the case of MARPLE-DPD.

Ferenc et al. (2005), in their Columbus tool, use machine learning algorithms to filter false positives out of the results of a graph matching phase, trying to provide better precision to the overall output. The main differences with our approach are related to the modeling approach, the employed techniques and the usage of the confidence values of the classification algorithm. The techniques they employed are C4.5 decision trees and neural networks with back propagation. In our approach, instead, the focus is more on SVMs, but some other common algorithms (including C4.5) have been experimented. Another difference is that MARPLE-DPD reports the confidence value produced by the classifier to the user, enabling a ranking of the reported pattern instances.

Another approach based on machine learning is the one from Guéhéneuc et al. (2004), where the authors use standard metrics as features for the machine learning process. The two main differences with our approach are that machine learning is not employed as a filter and the modeling phase is different. Their approach is based on a single detection step, where, for each class of the system, a set of metrics are used for learning and validation by a rule learner. The

Table 8
Main characteristics of DPD tools.

Tool	Language	Export format	Update	T	R
Columbus (Ferenc et al., 2005)	C++	n/a	n/a	S	A
CroCoPat (Beyer et al., 2003)	Java	RSF	02/2008	S	E
D-CUBED (Stencel and Wegrzynowicz, 2008)	Java	n/a	n/a	S	E
DP++ (Bansiya, 1998)	C++	n/a	n/a	S	E
Detection of patterns by joining forces (DPJF) (Binun and Kniesel, 2012)	Java	Prolog	11/2011	S	E
SSA (Tsantalis et al., 2006)	Java	XML	05/2010	S	A
DP-Miner (Dong et al., 2007)	Java	XML	01/2010	S	E
Design pattern recovery environment (DPRE; De Lucia et al., 2009)	C++, Java	n/a	n/a	S	E
DPVK (Wang and Tzerpos, 2005)	Eiffel	n/a	n/a	D	E
ePAD (De Lucia et al., 2010)	Java	Not supported	10/2010	D	E
Reclipse (Wendehals, 2003)	Java	Not supported	10/2011	D	A
Hedgehog (Blewitt et al., 2001)	Java	n/a	n/a	S	E
JBOORET (Hong et al., 2001)	C++	n/a	n/a	S	E
mb-pde (Birkner, 2007)	Java	XML	03/2008	D	E
Pat (Kramer and Prechelt, 1996)	C++	n/a	n/a	S	E
PINOT (Shi and Olsson, 2006)	Java	Unstructured text	10/2004	S	E
design motif identification multilayered approach (DeMIMA) (Guéhéneuc and Antoniol, 2008)	Java	Semi-structured text	04/2006	S	E
RSA	Java	XML	11/2011	S	E
SPOOL (Keller et al., 1999)	C++	n/a	n/a	S	E
SPQR (Smith and Stotts, 2003)	C++	XML	n/a	S	E
WoP (Dietrich and Elgar, 2007)	Java	XML	1.4.3	S	E
F.T. (Rasool and Mäder, 2011)	C, C++, C#, Java	n/a	n/a	S	E

R: recognition(A: approx, E: exact); T: technique(S: static, D: dynamic).

subjects of the classification are the classes, and not the pattern instances as in our approach. Guéhéneuc and Antoniol (2008) then developed the DeMIMA tool, which does not rely on machine learning, but exploits a constraint solver to perform an exact detection. In DeMIMA, on top of the constraint solver, an interactive procedure asks the user (if he wants) to remove one or more constraints, relaxing the constraints and including more results, which are imperfect design pattern instances. The amount of relaxation is used by the tool to give a score to the pattern instance. In a more recent work (Guéhéneuc et al., 2010), the two approaches were combined and tested, trying to achieve better performances both in terms of correctness and speed. The combination is achieved by first filtering classes using the fingerprinting approach, and then applying DeMIMA only on the filtered classes.

A different approach is the one defined by Dong et al. (2008). They use a compound record clustering algorithm to represent patterns as feature vectors, and then they apply a decision tree classifier. The clustering is applied to reduce the search space and to make the application of the classifier a feasible task. Design pattern vectors are represented similarly to single-level patterns in our approach. Their approach cannot represent multi-level patterns, and it is targeted to the usage of a small number of features, while our approach benefits of the information coming from micro-structures.

Tsantalis et al. (2006) use similarity scoring among matrices to allow partial matching between the definition of a pattern and the reported instances. Similarly, the approach implemented in Fujaba by Niere et al. (2002) is to give to each matching subpattern (which can be seen as a kind of micro-structure) a fuzzy value, that will influence the score of the entire pattern. The user can give different fuzzy weights to different subpatterns to influence the overall matching score.

The similarity of the two previous approaches (Niere et al., 2002; Tsantalis et al., 2006) and the one of DeMIMA (Guéhéneuc and Antoniol, 2008), which are not based on machine learning, with the one described in this paper is that a matching score can be reported to the user, giving an estimation of the matching confidence. As the score is not calculated by a machine learning technique, its value depends on the tool, and not on the user. DeMIMA has higher interaction with the user than the other two approaches, and lets the user control the amount of approximation of the detection. In our approach, instead, the user interacts with the detector by supplying new pattern examples. An advantage of our approach is that the example set increases over time, and can be reused in different projects.

In Table 8 we report a list of tools for DPD and we outline if the tools exploit exact or approximated recognition and static or dynamic techniques. Moreover, the table compares the tools according to some generic characteristics, *i.e.*, supported languages, export format, and last update. Other recent DPD approaches exist in the literature, *e.g.*, Pettersson and Lowe (2007), Zhao et al. (2007), Hu and Sartipi (2008), Bernardi and Lucca (2010), Issaoui et al. (2012), Alhusain et al. (2013), Rao and Gupta (2013), Yu et al. (2013), Alnusair et al. (2014), and Uchiyama et al. (2014), but to the best of our knowledge they have not been implemented as a tool, known to be available freely, commercially or under specific request.

9. Conclusions and future work

In this paper, we described a machine learning methodology applied to the problem of design pattern detection, implemented in a tool called MARPLE-DPD. The methodology was experimented on five design patterns (Singleton, Adapter, Composite, Decorator, Factory Method). Three performance values obtained by 10 different learning algorithms, in 22 different combinations, have been reported. We applied our experiments on a dataset composed of pattern instances extracted from 10 projects having a total size of about 540 kLOC. The training set was composed of a total of 2794 pattern instances.

The experiment demonstrates that the application of machine learning allows a significant performance increase on four patterns. For the remaining pattern, *i.e.*, Composite, the performance increase is not significant.

In the experiments, we subdivided patterns in two groups. For Singleton and Adapter, we applied only classification models, obtaining good performances, i.e., more than 85% of accuracy and F1-measure. For the second group, containing Composite, Decorator and Factory Method, we applied a combination of clustering and classification models. In this group, only Factory Method reached accuracy and F1-measure values higher than 80%. Decorator reached 82% accuracy and 77% F1-measure, while Composite 81% and 56% respectively. From our evaluation, lower performances were due to the lack of available pattern instance examples, resulting in a dataset of insufficient size.

In our approach, a general detection rule is applied to extract design pattern candidates from software systems. Candidates can be manually classified as correct or incorrect, and a set of classified candidates can be used to train a classification algorithm. The trained algorithm can then classify new unseen candidates as correct or incorrect.

In this way, the detection results benefit from learned criteria, which cannot be formally expressed in an exact matching rule. Correct and incorrect examples can be added to the training set over time, supporting an incremental approach, which allows to refine the detection algorithm in the same way that, *e.g.*, mail spam filters learn users preferences during their use. Moreover, the detected patterns are accompanied by a confidence value, estimating the quality of the detection. The confidence value allows the user to have results reported with a rank, suggesting the inspection order. In the paper, we did not make experiments regarding the quality of the obtained ranking, but the obtained AUC values tell us that we reach good results also in that sense, similar to the quality of classification. AUC, in fact, is strongly tied with the ranking obtained using confidence values, because of the way the ROC curve is built. Its usage for measuring the quality of rankings is widely accepted.

While some features of the approach are taken directly by the machine learning area, *e.g.*, the incremental learning approach and the ranking of results, we provide a concrete method for applying machine learning in the area of DPD, solving different issues related to the modeling of design pattern instances.

The approach does not rely on a particular algorithm or set of algorithms, but it just focuses on the general formulation of supervised and unsupervised learning. The fact of being "algorithm-agnostic" brings modularity in the implementation of the detection process, allowing free plugging of different algorithms without the need of rewriting some kind of adaptation code. The input of the classification process is defined in terms of micro-structures, which is a modeling choice allowing the expression of many different concepts related to source code elements. The classification process is not tied to a particular set of micro-structures, allowing the experimentation of different and new micro-structures.

We experimented several classifier algorithms. All the tested models performed better than the baseline ZeroR (i.e., a priori) model, except for Composite. The experiments outlined that many algorithms achieve comparable performances, while in most cases the best performances are obtained exploiting SVMs, decision trees and random forests. On the clustering algorithms side, we can say that K-means works better than CLOPE for this task.

Future work will be focused, in first place, in the direction of increasing the number of tested design patterns and software systems. The smaller datasets (*e.g.*, Composite) will be augmented to better understand the nature of the experimented detection issues. Other efforts will be spent in the test of more advanced clustering algorithms. In fact, more algorithms were available in Weka than the tested ones, but they needed excessive time and/or memory for computation. In further experiments, we will port the machine learning computation to other software packages and we will compare the obtained results with the current ones.

Another future development regards the comparison of our tools with other tools developed and proposed in the literature, as those cited in Section 8. Comparing tools is a long and complex task. In an accurate comparison, every reported instance must be identified as correct or incorrect (which can be a non-trivial task in itself), and then it must be matched against all the other instances reported by all the other tools, to understand if they represent the same pattern instance or not. This matching is time-consuming, because every tool and approach has its internal model of the reported pattern, producing different aggregation of instances and different way of reporting them. Even considering a small number of instances for each pattern, the number of comparisons is high, creating a serious problem in the scalability of this kind of comparison. Moreover, as opposed to pattern labeling, which is intrinsically subjective, a full comparison of different tools should be performed on a set of pattern instances having a shared and agreed evaluation. We are not aware of any data set having this characteristics.

In past work, we created the DPB (Arcelli Fontana et al., 2012), based on a simple and formal design pattern representation model. DPB contains and exposes all the results we gathered during the experiments reported in this paper. A different project with similar aims is DPDX (Kniesel et al., 2010), which defines a representation model for design patterns. DPDX is defined by major researchers in the DPD area, meaning that the need for a way of exchanging and comparing data in this field is widely known. We aim to collaborate with other researchers to enhance our platform and to create a *gold standard* to be used in the difficult task of tools comparison.

Appendix A. Design pattern definitions

In this section, some details regarding the way design patterns have been modeled and detected is reported. For each pattern, the micro-structures exploited by the Joiner and the merge rule (the data structure containing the pattern) are reported. Match rules are not reported, as they can be very long. The complete reference for match rules can be found in Zanoni (2012).

The micro-structures we report for each pattern are the ones exploited by the Joiner in the match phase. The way these micro-structures are combined can be expressed only by reporting complete match rules, which are available from Zanoni (2012). The micro-structures summary reported in this section has the purpose of detailing how many micro-structures are exploited by the Joiner in our setup, with respect to the full set exploited by the Classifier, which amounts to 86 micro-structures. The definitions of the reported micro-structures can be taken from different sources, depending on their type. EDPs were defined in Smith and Stotts (2002), and micro patterns in Gil and Maman (2005). Design pattern clues (clues in tables) have been published in Arcelli Fontana et al. (2011b). A comprehensive reference for all the micro-structures used in this work is reported anyway in Zanoni (2012). Most micro-structures used for match rules are the simplest ones. For example, EDPs are used for catching method invocations among roles, and clues are used for focusing on classes or methods with particular modifiers, or for describing methods' return or parameter types. More complex clues and micro patterns are not used in match rules, because they describe code structures with a lower degree of generality. These other data can be of great use for the Classifier, which can extract statistical dependencies involving the remaining micro-structures.

The purpose of reporting the merge rules is, instead, to allow better understanding of the way we define single-level and multi-level patterns. They also detail in a compact way which roles we considered in our pattern modeling and in which relation they are.

A.1. Singleton

Singleton is the simplest pattern, from the modeling point of view. It is composed of a single role, as reported in Listing A.4. The only role, *Singleton*, is contained in the only level defined, the root one. In fact, we defined Singleton as a single-level pattern. The micro-structures used for its matching, reported in Table A.1, are mainly related to object instantiation and to different ways of protecting attributes from being modified and constructors from being invoked.

```
<pattern name="Singleton">
  <role name="Singleton" />
</pattern>
```

Listing A.4. Merge rule for Singleton.

```
<pattern name="Adapter">
  <role name="Target" />
  <role name="Adapter" />
  <role name="Adaptee" />
</pattern>
```

Listing A.5. Merge rule for Adapter.

A.2. Adapter

Adapter is another single-level pattern. The merge rule reported in Listing A.5 defines three roles, all contained in the root level. This means that every possible assignment of the three roles to classes generates a new pattern instance. The exploited micro-structures, reported in Table A.2, address concepts like method invocation (through EDPs) and different variants of attribute protection, plus inheritance relationships.

A.3. Composite

Composite is defined as a multi-level design pattern. Its merge rule, reported in Listing A.6, assigns the three available roles to three different levels, two of which nested in the remaining one, representing the root. In other words, many *Composite* classes and many *Leaf* classes can occur in the pattern, which is identified solely by the class covering the *Component* role. The number of *Composite* and *Leaf* classes is independent, so they are kept in separate sublevels. The dependency defined among levels represents the inheritance relationship among the respective roles defined in the pattern definition. In fact, in Table A.3, we can see that the match phase addresses concepts like inheritance, abstractness and particular kinds of method invocation.

```
<pattern name="Composite">
  <role name="Component" />
  <sublevel>
    <role name="Composite" />
  </sublevel>
  <sublevel>
    <role name="Leaf" />
  </sublevel>
</pattern>
```

Listing A.6. Merge rule for Composite.

Table A.3
Micro-structures used in Composite's detection rule.

Micro-structure	Type
AbstractClass	Clue
ExtendedInheritance	Clue
Interface	Clue
RedirectInFamily	EDP
SameClass	Clue

Table A.1
Micro-structures used in Singleton's detection rule.

Micro-structure	Type
CreateObject	EDP
PrivateStaticReference	Clue
PrivateConstructor	Clue
ProtectedStaticReference	Clue
OtherStaticReference	Clue
PrivateConstructor	Clue
StaticFlag	Clue
ControlledException	Clue
ControlledInstantiation	Clue

Table A.2
Micro-structures used in Adapter's detection rule.

Micro-structure	Type
Conglomeration	EDP
CreateObject	EDP
Delegate	EDP
DelegatedConglomeration	EDP
DelegateInFamily	EDP
DelegateInLimitedFamily	EDP
ExtendedInheritance	EDP
ExtendMethod	EDP
Interface	Clue
Joiner	Micro pattern
OtherInstanceReference	Clue
OtherStaticReference	Clue
PrivateInstanceReference	Clue
PrivateStaticReference	Clue
ProductReturns	Clue
ProtectedInstanceReference	Clue
ProtectedStaticReference	Clue
ReceivesParameter	Clue
Recursion	EDP
Redirect	EDP
RedirectInFamily	EDP
RedirectInLimitedFamily	EDP
RedirectRecursion	EDP
RevertMethod	EDP
SameClass	Clue

A.4. Decorator

Decorator is another multi-level pattern. It defines four roles in four different levels, with two levels of nesting. For this pattern, the choice we make is to consider a single pattern instance all the possible decorators of a *Component* (Listing A.7). This choice has been made to allow recovering a wider picture of the recovered decorators, and allow better understanding of the options available when decorating a *Component*. The concepts considered by the matching rule, represented by the micro-structures reported in Table A.4, are related to particular kinds of method invocations, to inheritance and abstract types.

Table A.4
Micro-structures used in Decorator's detection rule.

Micro-structure	Type
AbstractType	Clue
Conglomeration	EDP
Delegate	EDP
DelegateInFamily	EDP
DelegateInLimitedFamily	EDP
DelegatedConglomeration	EDP
ExtendMethod	EDP
ExtendedInheritance	Clue
Recursion	EDP
Redirect	EDP
RedirectInFamily	EDP
RedirectInLimitedFamily	EDP
RedirectRecursion	EDP
RevertMethod	EDP
SameClass	Clue

```
<pattern name="Decorator">
  <role name="Component" />
  <sublevel>
    <role name="Decorator" />
    <sublevel>
      <role name="ConcreteDecorator" />
    </sublevel>
  </sublevel>
  <sublevel>
    <role name="ConcreteComponent" />
  </sublevel>
</pattern>
```

Listing A.7. Merge rule for Decorator.

```
<pattern name="FactoryMethod">
  <role name="Creator" />
  <role name="ConcreteCreator" />
  <role name="Product" />
  <sublevel>
    <role name="ConcreteProduct" />
  </sublevel>
</pattern>
```

Listing A.8. Merge rule for Factory Method.

Table A.5
Micro-structures used in Factory Method's detection rule.

Micro-structure	Type
CreateObject	EDP
ExtendedInheritance	Clue
ProductReturns	Clue
SameClass	Clue

A.5. Factory method

Factory Method is a multi-level pattern, with a simpler structure than Composite and Decorator. In fact, Listing A.8 defines that three roles are in the root level, defining the pattern instance identity, and the *ConcreteProduct* role is in a nested level, telling that many *Product* subclasses can be created by the *ConcreteCreator*. The choice of keeping most roles in the root level is due to the need of catching real instances of the pattern, which tend to deviate significantly from the original definition. In fact, the original definition of Factory Method is structurally very similar to Abstract Factory, while having different purposes and complexity. Many Factory Method implementations have little structure, and we could better catch those variants using this modeling choice. As for the exploited micro-structures, reported in Table A.5 only a few concepts were used, i.e., inheritance, object creation and methods return types.

References

Alhusain, S., Coupland, S., John, R., Kavanagh, M., 2013. Towards machine learning based design pattern recognition. In: 13th UK Workshop on Computational Intelligence (UKCI'13), IEEE, Guildford, UK, pp. 244–251. doi:10.1109/UKCI.2013.6651312

Alnusair, A., Zhao, T., Yan, G., 2014. Rule-based detection of design patterns in program code. Int. J. Softw. Tool Technol. Transf. 16, 315–334. doi:10.1007/s10009-013-0292-z

Arcelli Fontana, F., Caracciolo, A., Zanoni, M., 2012. DPB: A benchmark for design pattern detection tools. In: Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR'12). IEEE Computer Society, Szeged, Hungary, pp. 235–244. doi:10.1109/CSMR.2012.32

Arcelli Fontana, F., Maggioni, S., Raibulet, C., 2011. Design patterns: A survey on their micro-structures. J. Softw. Maint. Evol.: Res. Pract. 25, 27–52. doi:10.1002/smr.547

Arcelli Fontana, F., Zanoni, M., 2011. A tool for design pattern detection and software architecture reconstruction. Inform. Sci. 181, 1306–1324. doi:10.1016/j.ins.2010.12.002

Arcelli Fontana, F., Zanoni, M., Maggioni, S., 2011. Using design pattern clues to improve the precision of design pattern detection tools. J. Object Technol. 10, 4:1–31. doi:10.5381/jot.2011.10.1.a4

Ball, T., 1999. The concept of dynamic analysis. In: Nierstrasz, O., Lemoine, M. (Eds.), Software Engineering (ESEC/FSE'99). *Lecture Notes in Computer Science*, vol. 1687. Springer, Berlin/Heidelberg, pp. 216–234. doi:10.1007/3-540-48166-4_14

Bansiya, J., 1998. Automating design-pattern identification. Dr Dobbs J. http://drdobbs.com/architecture-and-design/184410578.

Bayley, I., Zhu, H., 2010. Formal specification of the variants and behavioural features of design patterns. J. Syst. Softw. 83, 209–221. doi:10.1016/j.jss.2009.09.039

Bernardi, M.L., Lucca, G.A., 2010. Mining design patterns in object oriented systems by a model-driven approach. In: Kim, T., Kim, H., Khan, M., Kiumi, A., Fang, W., Ślęzak, D. (Eds.), Advances in Software Engineering. *Communications in Computer and Information Science*, vol. 117. Springer, Berlin/Heidelberg, pp. 67–77. 10.1007/978-3-642-17578-7_8

Beyer, D., Noack, A., Lewerentz, C., 2003. Simple and efficient relational querying of software structures. In: Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03). IEEE Computer Society, Victoria, B.C., Canada, pp. 216–225. 10.1109/WCRE.2003.1287252.

Binun, A., Kniesel, G., 2012. DPJF—Design pattern detection with high accuracy. In: Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR 2012). IEEE Computer Society, Szeged, Hungary, pp. 245–254. doi:10.1109/CSMR.2012.82

Birkner, M., 2007. Objected-oriented Design Pattern Detection Using Static and Dynamic Analysis in Java Software (Master's thesis). Sankt Augustin, Germany: Bonn-Rhine-Sieg University of Applied Sciences. http://mb-pde.googlecode.com/files/MasterThesis.pdf.

Blewitt, A., Bundy, A., Stark, I., 2001. Automatic verification of java design patterns. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE 2001). IEEE Computer Society, San Diego, CA, USA, pp. 324–327. doi:10.1109/ASE.2001.989821

De Lucia, A., Deufemia, V., Gravino, C., Risi, M., 2009. Design pattern recovery through visual language parsing and source code analysis. J. Syst. Softw. 82, 1177–1193. doi:10.1016/j.jss.2009.02.012

De Lucia, A., Deufemia, V., Gravino, C., Risi, M., 2010. An eclipse plug-in for the detection of design pattern instances through static and dynamic analysis. In: Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM'10). IEEE Computer Society, Timisoara, Romania, pp. 1–6. doi:10.1109/ICSM.2010.5609707

Dietrich, J., Elgar, C., 2007. Towards a web of patterns. Web Semant.: Sci. Service Agent World Wide Web 5, 108–116. doi:10.1016/j.websem.2006.11.007

Dong, J., Lad, D.S., Zhao, Y., 2007. Dp-miner: Design pattern discovery using matrix. In: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS 2007). IEEE Computer Society, Tucson, Arizona, USA, pp. 371–380. doi:10.1109/ECBS.2007.33

Dong, J., Sun, Y., Zhao, Y., 2008. Compound record clustering algorithm for design pattern detection by decision tree learning. In: IEEE International Conference on Information Reuse and Integration (IRI'08). IEEE Computer Society, Las Vegas, NV, USA, pp. 226–231. doi:10.1109/IRI.2008.4583034

Ferenc, R., Beszédes, A., Fülöp, L., Lele, J., 2005. Design pattern mining enhanced by machine learning. In: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05). IEEE Computer Society, Budapest, Hungary, pp. 295–304. doi:10.1109/ICSM.2005.40

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Professional.

Gil, J.Y., Maman, I., 2005. Micro patterns in Java code. In: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA 2005). ACM, San Diego, CA, USA, pp. 97–116. doi:10.1145/1094811.1094819

Guéhéneuc, Y.G., 2007. PMARt: Pattern-like micro architecture repository. In: Weiss, M., Birukou, A., Giorgini, P. (Eds.), Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories. http://www.patternforge.net/wiki/images/e/e6/Gueheneuc.pdf.

Guéhéneuc, Y.G., Antoniol, G., 2008. DeMIMA: A multilayered approach for design pattern identification. IEEE Transactions on Software Engineering 34, 667–684. doi:10.1109/TSE.2008.48

Guéhéneuc, Y.G., Guyomarc'h, J.Y., Sahraoui, H., 2010. Improving design-pattern identification: A new approach and an exploratory study. Softw. Qual. J. 18, 145–174. doi:10.1007/s11219-009-9082-y

Guéhéneuc, Y.G., Sahraoui, H., Zaidi, F., 2004. Fingerprinting design patterns. In: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04). IEEE Computer Society, Victoria, BC, Canada, pp. 172–181. doi:10.1109/WCRE.2004.21

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., 2009. The weka data mining software: An update. SIGKDD Explor. Newsl. 11, 10–18. doi:10.1145/1656274.1656278

Hong, M., Xie, T., Yang, F., 2001. JBOORET: An automated tool to recover oo design and source models. In: Proceedings of the 25th Annual International Computer Software and Applications Conference (COMPSAC'01). IEEE Computer Society, Chicago, IL, USA, pp. 71–76. doi:10.1109/CMPSAC.2001.960600

Hu, L., Sartipi, K., 2008. Dynamic analysis and design pattern detection in java programs. In: Proceedings of the 20th International Conference on Software Engineering & Knowledge Engineering (SEKE 2008). Knowledge Systems Institute, Illinois, 842–846.

Issaoui, I., Bouassida, N., Ben-Abdallah, H., 2012. A design pattern detection approach based on semantics. In: Lee, R. (Ed.), Software Engineering Research, Management and Applications. *Studies in Computational Intelligence*, vol. 430. Springer, Berlin Heidelberg, pp. 49–63. doi:10.1007/978-3-642-30460-6_4

Keller, R.K., Schauer, R., Robitaille, S., Pagé, P., 1999. Pattern-based reverse-engineering of design components. In: Proceedings of the 21st International Conference on Software Engineering (ICSE'99). ACM, Los Angeles, CA, USA, pp. 226–235. doi:10.1145/302405.302622

Kniesel, G., Binun, A., Hegedüs, P., Fülöp, L.J., Chatzigeorgiou, A., Guéhénéuc, Y.G., Tsantalis, N., 2010. DPDX—Towards a common result exchange format for design pattern detection tools. In: Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR'10). IEEE Computer Society, Madrid, Spain, pp. 232–235. doi:10.1109/CSMR.2010.40

Kramer, C., Prechelt, L., 1996. Design recovery by automated search for structural design patterns in object-oriented software. In: Proceedings of the Third Working Conference on Reverse Engineering (WCRE'06). IEEE Computer Society, Monterey, CA, USA, pp. 208–215. doi:10.1109/WCRE.1996.558905

Marlowe, T.J., Ryder, B.G., 1990. Properties of data flow frameworks. Acta Informat. 28, 121–163. 10.1007/BF01237234

Niere, J., Schäfer, W., Wadsack, J.P., Wendehals, L., Welsh, J., 2002. Towards pattern-based design recovery. In: Proceedings of the 24th International Conference on Software Engineering (ICSE'02). ACM, Orlando, FL, USA, pp. 338–348. doi:10.1145/581339.581382

Pettersson, N., Lowe, W., 2007. A non-conservative approach to software pattern detection. In: 15th IEEE International Conference on Program Comprehension (ICPC'07). IEEE Computer Society, pp. 189–198. doi:10.1109/ICPC.2007.8

Pettersson, N., Löwe, W., Nivre, J., 2010. Evaluation of accuracy in design pattern occurrence detection. IEEE Trans. Softw. Eng. 36, 575–590. doi:10.1109/TSE.2009.92

Rao, R.S., Gupta, M., 2013. Design pattern detection by a heuristic graph comparison algorithm. Int. J. Adv. Res. Comput. Sci. Softw. Eng. 3, 251–255. http://www.ijarcsse.com/docs/papers/Volume_3/11_November2013/V3I11-0229.pdf.

Rasool, G., Mäder, P., 2011. Flexible design pattern detection based on feature types. In: Proceedings of 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11). IEEE Computer Society, Lawrence, KS, USA, pp. 243–252. doi:10.1109/ASE.2011.6100060

Samadzadegan, F., Soleymani, A., Abbaspour, R.A., 2010. Evaluation of genetic algorithms for tuning svm parameters in multi-class problems. In: Proceedings of the 11th International Symposium on Computational Intelligence and Informatics (CINTI'10). IEEE Computer Society, Budapest, Hungary, pp. 323–328. doi:10.1109/CINTI.2010.5672224

Shi, N., Olsson, R.A., 2006. Reverse engineering of design patterns from java source code. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06). IEEE Computer Society, Tokyo, Japan, pp. 123–134. doi:10.1109/ASE.2006.57

Smith, J.M., Stotts, D., 2002. Elemental Design Patterns: A formal semantics for composition of OO software architecture. In: Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02). IEEE Computer Society, Greenbelt, Maryland, USA, pp. 183–190. doi:10.1109/SEW.2002.1199472

Smith, J.M., Stotts, D., 2003. SPQR: Flexible automated design pattern extraction from source code. In: Proceedings of the 18st IEEE/ACM International Conference on Automated Software Engineering (ASE'03). IEEE Computer Society, Montreal, Canada, pp. 215–224. doi:10.1109/ASE.2003.1240309

Software Engineering Research Center, 2012. Research projects: Design patterns. http://research.ciitlahore.edu.pk/Groups/SERC/DesignPatterns.aspx.

Stencel, K., Wegrzynowicz, P., 2008. Detection of diverse design pattern variants, in: Proceedings of the 15th Asia-Pacific Software Engineering Conference (APSEC'08), IEEE Computer Society, Beijing, China. pp. 25–32. doi:10.1109/APSEC.2008.67.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.T., 2006. Design pattern detection using similarity scoring. IEEE Trans. Softw. Eng. 32, 896–909. doi:10.1109/TSE.2006.112

Uchiyama, S., Kubo, A., Washizaki, H., Fukazawa, Y., 2014. Detecting design patterns in object-oriented program source code by using metrics and machine learning. J. Softw. Eng. Appl. 7, 983–998. doi:10.4236/jsea.2014.712086

Wang, W., Tzerpos, V., 2005. Design pattern detection in eiffel systems. In: Proceedings of the 12th Working Conference on Reverse Engineering (WCRE'05). IEEE Computer Society, Pittsburgh, Pennsylvania, USA, pp. 165–174. doi:10.1109/WCRE.2005.14

Wendehals, L., 2003. Improving design pattern instance recognition by dynamic analysis. In: Proceedings of the ICSE 2003 Workshop on Dynamic Analysis (WODA'03). Jonathan Cook, New Mexico State University, Portland, Oregon, USA, pp. 29–32. http://www.cs.nmsu.edu/~jcook/woda2003/woda2003.pdf

Witten, I.H., Frank, E., Hall, M.A., 2011. Data Mining: Practical Machine Learning Tools and Techniques, third ed. Morgan Kaufmann.

Yang, Y., Guan, X., You, J., 2002. CLOPE: A fast and effective clustering algorithm for transactional data. In: Proceedings of the eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'02). ACM, Edmonton, Alberta, Canada, pp. 682–687. doi:10.1145/775047.775149

Yu, D., Zhang, Y., Ge, J., Wu, W., 2013. From sub-patterns to patterns: An approach to the detection of structural design pattern instances by subgraph mining and merging. In: Proceedings of the 37th Annual Computer Software and Applications Conference (COMPSAC'13), IEEE, Kyoto, Japan, pp. 579–588. doi:10.1109/COMPSAC.2013.92

Zanoni, M., 2012. Data mining techniques for design pattern detection. (Ph.D. thesis). Milano, Italy: Università degli Studi di Milano-Bicocca, Dottorato di ricerca in INFORMATICA. http://hdl.handle.net/10281/31515.

Zhao, C., Kong, J., Dong, J., Zhang, K., 2007. Pattern-based design evolution using graph transformation. J. Vis. Lang. Comput. 18, 378–398. [Special issue: Visual Interactions in Software Artifacts]. http://dx.doi.org/10.1016/j.jvlc.2007.07.004

Marco Zanoni received the M.S. and Ph.D. degrees in Computer Science from the University of Milano-Bicocca. He is a post-doc research fellow of the Department of Informatics, Systems and Communication of the University of Milano-Bicocca. His research interests include design pattern detection, software architecture reconstruction and machine learning.

Francesca Arcelli Fontana received the Diploma and Ph.D. degrees in Computer Science from the University of Milano. She was at University of Salerno and University of Sannio, Faculty of Engineering, as assistant professor in software engineering. She is currently an associate professor in the Department of Computer Science of the University of Milano-Bicocca. Her research interests include software evolution, software metrics, software maintenance and reverse engineering. She is a member of the IEEE and the IEEE Computer Society.

Fabio Stella graduated in Computer Sciences in 1991 at the University of Milano. From 1991 to 1994 he worked as research assistant for the EEC IMPROD project on semiconductors failure diagnosis, analysis and quality improvement. In 1994 he became assistant professor of Operations Research at the University of Milano. He received the Ph.D. in Computational Mathematics and Operations Research in 1995. In 2001 he became associate professor of Operations Research at the University of Milano-Bicocca. He directs the Models and Algorithms for Data and Text Mining Laboratory and actively collaborates with many Italian SMEs in the area of document management, financial risk management, and analysis of clinical and microarray data. His main research interests are; data mining, text mining, and computational finance.