The Journal of Systems and Software 103 (2015) 102–117

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

ARTICLE INFO

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

ABSTRACT

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.