GEML: A Grammar-based Evolutionary Machine Learning Approach for Design-Pattern Detection

Rafael Barbudo[a], Aurora Ramírez[a], Francisco Servant[b], José Raúl Romero[a,*]

[a] *Department of Computer Science and Numerical Analysis, University of Córdoba, 14071, Córdoba, Spain*
[b] *Department of Computer Science, Virginia Tech, VA 24060, Blacksburg, USA*

Abstract

Design patterns (DPs) are recognised as a good practice in software development. However, the lack of appropriate documentation often hampers traceability, and their benefits are blurred among thousands of lines of code. Automatic methods for DP detection have become relevant but are usually based on the rigid analysis of either software metrics or specific properties of the source code. We propose GEML, a novel detection approach based on evolutionary machine learning using software properties of diverse nature. Firstly, GEML makes use of an evolutionary algorithm to extract those characteristics that better describe the DP, formulated in terms of human-readable rules, whose syntax is conformant with a context-free grammar. Secondly, a rule-based classifier is built to predict whether new code contains a hidden DP implementation. GEML has been validated over five DPs taken from a public repository recurrently adopted by machine learning studies. Then, we increase this number up to 15 diverse DPs, showing its effectiveness and robustness in terms of detection capability. An initial parameter study served to tune a parameter setup whose performance guarantees the general applicability of this approach without the need to adjust complex parameters to a specific pattern. Finally, a demonstration tool is also provided.

Keywords:
design pattern detection, reverse engineering, machine learning, associative classification, grammar-guided genetic programming

1. Introduction

Design patterns (DPs) are reusable template solutions that address recurrent software-design problems. The adoption of DPs is a best practice for programmers with the goal of improving the quality of software products — in terms of their maintainability, elegance, flexibility and understandability (Gamma et al., 1995). Given such benefits, and considering that manual inspection is an error-prone and time-consuming process, automatic Design Pattern Detection (DPD) has become a prominent area in the reverse-engineering research field (Bafandeh Mayvan et al., 2017). By automatically identifying the adoption of DPs, DPD techniques can improve the understanding of the design decisions in software systems, as well as the processes of redocumenting them, reimplementing them, and reusing them.

Different techniques have been proposed in the research literature to automate DPD, most of which search for particular structures in static code (Mayvan and Rasoolzadegan, 2017). In this case, the code structures defining DPs need to be predetermined by experts into a knowledge-base, as they usually are specific to the codebase of study. Having the expert defined these code structures may impose rigidity to the detection technique, and design patterns should be reinterpreted for particular contexts.

To reduce this limitation, machine learning (ML) techniques were proposed for DPD, *e.g.,* Ferenc et al. (2005). These techniques are more easily adapted to different codebases because they learn from a collection of representative examples — and thus can recognise diverse implementations by simply replacing or extending such analysed collection of examples. In particular, ML-based approaches provide mechanisms to learn the structural and behavioural properties of the source code, as well as software metrics, that best describe the DP. Even so, despite the fact that DPs can be described considering these multiple aspects, current ML approaches consider either software metrics or code properties — structural, behavioural or both.

Unfortunately, the detection performance of existing ML-based approaches for DPD is affected by the design pattern under analysis, often requiring its specific parameter configuration. This characteristic makes them harder to adopt in practice: tuning parameters requires considerable effort by software engineers, since machine learning is often not their main area of expertise. Furthermore, the

*Corresponding author. Tel. +34 957 21 26 60
Email addresses: rbarbudo@uco.es (Rafael Barbudo), aramirez@uco.es (Aurora Ramírez), fservant@vt.edu (Francisco Servant), jrromero@uco.es (José Raúl Romero)

results provided by certain ML-based techniques like neural networks are difficult to understand and interpret by the human expert (Mori and Uchihira, 2019), thus making their recommendations less likely to be trusted (Dzindolet et al., 2003).

In this paper, we present GEML, a novel machine-learning-based approach for DPD from static code. More specifically, GEML is founded on associative classification (AC), a ML technique for which we have implemented a solution based on grammar-guided genetic programming (G3P). Our goal is to achieve the benefits of ML-based proposals while also addressing their limitations by using G3P as a basis of our approach. Like other ML-based proposals, GEML learns from DP examples, giving it the ability to capture diverse DP implementations. As for the limitations of previous ML proposals, GEML has been designed to promote extensibility, readability and flexibility.

For extensibility, GEML includes a customisable collection of design microstructures whose potential to identify DP instances is automatically determined by the evolutionary algorithm during the learning phase. For readability, GEML builds a rule-based classifier guided by a configurable context-free grammar that declares the syntax of the rules. Rules, as a well-established mechanism to encode human knowledge (Grosan and Abraham, 2011), describe the distinctive characteristics of DP instances in a more comprehensible way than the outcomes produced by black-box models (Kotsiantis et al., 2006). And for flexibility, GEML is applicable to each DP without requiring different algorithms or parameter configurations, since it searches for optimal rules to describe each particular DP.

In addition to these benefits, in our experiments, GEML also provided accuracy levels that outperformed other available techniques, providing competitive results even when very few samples are provided. Furthermore, GEML is able to maintain a stable behaviour with just a single general configuration, which we believe will make it more beneficial for software engineers in practical settings.

In short, GEML allows developers to execute it out-of-the-box without having to configure it for each DP, and customise its elements and parameters to gain detection power in particular scenarios. However, these desirable qualities would only make GEML beneficial in practice if it also provided (reasonably) similar performance as other existing techniques. Thus, we focus our evaluation on studying the effectiveness provided by GEML.

We perform several experiments in our evaluation. Firstly, we study GEML's detection performance in depth through a sensitivity analysis. Although these kinds of analysis are laborious, they become essential to comprehend the internals of artificial intelligence techniques like those applied here. In particular, we aim to understand the extent to which each algorithmic component and its configuration contributed to detecting each individual DP. In this way we are able to determine the best detection capability offered by GEML. We also study the selection options and effectiveness of GEML if its configuration was customised in terms of the design microstructures eligible by the software engineer and software metrics — called operators — that describe the properties that best characterise each DP. Then, we also experimented with the alternative scenario in which software engineers did have the knowledge and time to customise it separately for each DP. As additional benefits of customising GEML individually for each DP, we expect that practitioners would also observe lower runtimes. Finally, we measure the detection performance of the general configuration of GEML. We anticipate that this would be the most common (and simpler) usage scenario for practitioners.

We also analyse how GEML behaves when the general configuration is set but the training conditions change. In this case, we consider up to 15 DPs — the highest number of DPs studied from a ML perspective — at the cost of reducing the number of available training samples. Even so, our results show that GEML is able to infer detection rules in all cases, not requiring any adaptation regarding its internal components and configuration. We also compare GEML's detection performance to other DPD methods, including both ML and non-ML techniques. GEML provided higher accuracy and F_1 score than MARPLE (Zanoni et al., 2015) — a well-known ML-based approach — for four out of the five DPs available for comparison (with up to 35% improvement in F_1 for one of them). Against other non-ML-based techniques, GEML also recover more DP instances when validating with JHotDraw, a frequently studied project in DPD literature. Lastly, we analyse the strengths and weaknesses of GEML with respect to SSA and Ptidej, the two reference DPD tools most frequently used for comparative purposes. In this sense, GEML is highly competitive since it correctly identifies a higher number of DP implementations and support DPs whose detection is not available in these tools. GEML also overcomes some limitations of these tools, such as the absence of the classes implementing some roles of the DP and the excessive number of false positives. In short, the results of our evaluation show that GEML is a practical approach for DPD: it improves the effectiveness of current methods, while returning readable outcomes. Furthermore, the possibility of choosing the code properties more relevant for learning brings flexibility to the DPD process.

This paper provides the following contributions:

- a novel DPD technique (GEML) based on machine learning with grammar-guided genetic programming that is able to provide a single configuration for detecting 15 DPs and returning human-readable rules;
- the provision of a collection of design microstructures and metrics, in terms of operators that are derived by the context-free grammar (CFG) guiding the G3P algorithm and defined with both categorical and numerical values;
- an analysis of which design microstructures and soft-

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

ware metrics best represent each DP.

- a experimental evaluation of GEML under different training and configuration scenarios, finding highly competitive results when compared against other ML and non-ML approaches;

- a research demonstration tool to support software engineers in executing GEML and customising it for individual DPs.

The rest of the paper is organised as follows. Main concepts and terminology related to the applied techniques are introduced in Section 2. Section 3 presents the related work, and Section 4 provides a detailed description of our DPD model. Section 5 details the experimental methodology and framework. Then, the three experiments conducted are discussed in Section 6, 7 and 8. Section 9 describes the demonstration tool provided as additional material supporting this approach. Finally, threats to validity and concluding remarks are presented in Sections 10 and 11, respectively.

2. Theoretical Background

Design patterns are descriptions of communicating objects and classes that are customised to solve a general design problem in a particular context (Gamma et al., 1995). They differ in the number and purpose of their defining roles, each one describing a specific task to be performed. Consequently, a DPD method does not only identify the code elements implementing the design pattern, but also the roles they play within the pattern structure. Notice that the definition of these elements and how they relate to each other might depend on the particularities of the programming language, *e.g.,* Java allows implementing explicit interfaces but C++ does not. Besides, a given role could be played by more than one code element. Therefore, since design patterns are general and adaptable solutions, the presence of certain properties or a predefined structure for roles cannot be assumed. Their diverse nature, *i.e.,* behavioural, creational or structural, suggests that different properties might be needed to characterise a given pattern, what definitely may hamper the detection process.

This section explains the most relevant theoretical concepts related to the techniques required by GEML to conduct the detection procedure. This is built on the basis of associative classification, *i.e.,* a ML-based approach founded on the use of *if-then* rules over a classification approach looking for an understandable detection model (Thabtah, 2007). Internally, the classifier is constructed using an evolutionary technique, G3P, specially conceived to evolve computer programs according to a grammar, whose derivations and constraints define how these programs — *i.e.,* a potential rule of the GEML's detection model — are built.

2.1. Associative Classification

Machine learning provides computational methods that allow learning from vast collections of data. ML techniques can be mostly divided into two major groups, unsupervised and supervised. Unsupervised techniques explore the data samples to find interesting and meaningful patterns describing them. A well-known technique within unsupervised learning is association rule mining (ARM) (Agrawal et al., 1993), where patterns are represented as a set of association rules. Formally, let I = $\{i_1, ..., i_n\}$ be a set of items, and let A and C be itemsets, *i.e.,* A = $\{i_1, ..., i_j\}$ and C = $\{i_1, ..., i_k\}$, an association rule is an implication of the type $A \rightarrow C$ where A $\subset$ I, C $\subset$ I, and A $\cap$ C = $\emptyset$.

In ARM, quality measures are computed to determine the quality of the produced rules, the support and confidence being the most representative measures. On the one hand, the support (Eq. 1) indicates how frequently a rule is satisfied within a given set of data samples (D). On the other hand, the confidence (Eq. 2) measures the proportion of samples that satisfy the consequent from those already satisfying the antecedent. In both equations, s represents each data sample within D.

$$supp(A \rightarrow C) = \frac{|\{A \cup C \subseteq s\}|}{|D|} \quad s \in D \tag{1}$$

$$conf(A \rightarrow C) = \frac{|\{A \cup C \subseteq s\}|}{|\{A \subseteq s\}|} \quad s \in D \tag{2}$$

Unlike unsupervised learning, supervised techniques make predictions based on the information extracted from past data. The purpose of a classification algorithm is to assign a predefined category, referred as class, to an unknown data sample. In particular, associative classification makes use of ARM techniques to build rule-based classifiers (Thabtah, 2007). With this aim, class association rules (CARs) (Liu et al., 1998), those in which the consequent determines the class, are generated. In general, AC can be viewed as a two-step process: (1) an ARM-based method is applied to mine a set of CARs; and (2) the set of CARs is pruned to exclusively select those rules that will be definitely constitute the classification model.

2.2. Grammar-guided Genetic Programming (G3P)

Inspired by the principles of natural evolution, evolutionary computation creates a population of individuals, each one representing a potential solution to the problem, which are then "evolved" during a number of generations (Eiben and Smith, 2015). An individual is characterised by its genotype, *i.e.,* the computational structure used to encode the solution, and its phenotype, *i.e.,* its real-world representation. In addition, a domain-specific fitness function has to be defined to assess the quality of an individual. In a general schema, for each generation, individuals are selected and modified with the aim of producing better individuals than their predecessors. More

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

specifically, the algorithm selects a subset of the population to act as parents, often based on their fitness value. Parents are then combined to generate offspring by means of a crossover operator. Mutation can also be applied to the obtained individuals, looking for more diverse solutions. At the end of the generation, the best individuals survive and create the next population.

Genetic programming (GP) is a special type of EC technique in which individuals are encoded using tree structures (Koza, 1992). Originally conceived to evolve computer programs, GP requires specialised operators to manipulate tree-based solutions of different shapes and sizes. In particular, G3P is an extension of GP in which a CFG describes the syntactic constrains that must be satisfied by any valid individual. A CFG is defined by a four-tuple $\{S, \sum_N, \sum_T, P\}$, where S is the root symbol, $\sum_N$ is the set of non-terminal symbols, $\sum_T$, is the set of terminal symbols and P is the set of production rules. A production rule indicates how a non-terminal symbol can be rewritten into one of their derivations until the expression only contains terminal symbols. Formally, it can be expressed as $a \rightarrow B$, where $a \in \sum_N$ and $B \in \{\sum_N \cup \sum_T\}^*$.

In G3P, each individual is created by deriving a different sequence of production rules, represented as a derivation tree. The elements of the CFG are also considered during the application of crossover and mutation to guarantee the production of valid solutions. G3P has been used to mine association rules (Luna et al., 2012), where the grammar formally defines the structure of the rule in terms of itemsets. In this context, each individual represents an association rule, and the evolutionary process is oriented towards finding a set of high-quality rules according to support and confidence criteria.

3. Related Work

Multiple techniques have been proposed for automatic detection of structural, behavioural and creational design patterns, using both static and dynamic analysis of source code. Also, both types of techniques can be combined together depending on the pattern to be detected, *e.g.,* by inspecting both class definitions and object collaborations (Ng et al., 2010; Lucia et al., 2018). Given that GEML performs static analysis, we mostly focused on these approaches for DP detection, with special attention to those proposals based on machine learning. In addition, even though the majority of studies take source code as input, other authors have explored detection at design stages too, *e.g.,* using UML class diagrams (Di Martino and Esposito, 2016).

Within the scope of reasoning techniques, the Pat system (Kramer and Prechelt, 1996) is considered a pioneering work based on declarative logic programming. It defines the structural properties of DPs and the software project as Prolog facts. Then, the Prolog search engine is used to look for exact matches of these facts, which are identified as new DP instances. Similarly, the use of Prolog facts together with the use of meta-patterns (Pree and Sikora, 1997), which are common structures of DPs, were used by Hayashi et al. (2008). In this context, the FINDER tool (Dabain et al., 2015) integrates facts related to class structure and method invocations. Then, it filters out the fact base using predefined detection scripts. Fuzzy logic has been also used to deal with incomplete knowledge (Niere et al., 2002) in a attempt to make the detection process more flexible. In this approach, a fuzzy weight is assigned to each either structural or behavioural property thus reporting a level of confidence to each potential match. However, fuzzy-based approaches require an extensive base of knowledge to represent the diverse implementations. Given that such a base of knowledge has to be constructed by a group of experts, outcomes from the detection process could be biased.

Alternatively, techniques based on similarity scoring adopt graphs to represent structural information. The SSA tool (Tsantalis et al., 2006) searches for substructures that correspond to a predefined template of the graph describing the structure of a certain DP, *i.e.,* the explicit representation of the relationships between roles. Mechanisms to deal with approximate matches can be applied too, such as considering a small variation range in the value of some properties. Similarly, behavioural properties like message passing are often considered to reduce false positives (Dong et al., 2009). In a different approach (Yu et al., 2015), a set of substructures are searched prior to the identification of the DP implementations. Then, method signatures are compared against a set of predefined templates describing the DPs to refine the results. In this vein, two other state-of-the-art approaches focus on improving the prediction performance of the substructures search process (Bafandeh Mayvan et al., 2017; Yu et al., 2018). While the former reduces the search space by conveniently partitioning the project graph, the latter defines ordered sequences to guide the search in such an order that the most representative classes of the pattern are discovered first, filtering irrelevant classes at an early stage. DPF is a model-based graph matching tool for which a meta-model and a domain-specific language are proposed to specify the structural and behavioural relationships describing DPs (Bernardi et al., 2014). Another popular tool, PINOT (Shi and Olsson, 2006), uses graphs as an abstract representation of methods taking part in candidate DP implementations, from which control flow is statically analysed.

Formal methods have been explored in the context of DPD too. More specifically, formal concept analysis has been applied to find groups of classes sharing common structural relations that represent candidate design patterns (Tonella and Antoniol, 2001). This approach was later extended introducing some additional contributions like a filtering phase of candidate patterns and proposing a language-independent variant (Arevalo et al., 2004). A case study was also conducted to detect structural patterns within two subsystems of a printer controller (Wierda

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

et al., 2009). However, formal approaches can be computationally expensive due to the exponential growth of formal concepts (Poelmans et al., 2013).

Design patterns have also been expressed in terms of features and documented using annotations, from which automatic analysers can be built (Rasool et al., 2010; Rasool and Mader, 2011). As the authors acknowledge, incomplete definitions or inappropriate semantics negatively impact the detection process. Relations defined on the basis of a visual language represent another way to specify the properties of structural DPs (Lucia et al., 2009). In order to reduce false positives of the detection process, this method supports the definition of negative criteria, *i.e.,* those properties that do not indicate the presence of a DP. Then it performs a low-level analysis of the relationships between classes in a second step.

Other authors have formulated DPD as a constraint satisfaction problem, defining those structural and behavioural conditions that each particular DP should satisfy. This type of method requires the manual definition and formalisation of the relationships between roles. DeMIMA (Guéhéneuc and Antoniol, 2008) and the DPJF tool (Binun and Kniesel, 2012) are representative examples of this approach, both supporting constraint relaxation. The former, which provides explanations of satisfied and non-satisfied constraints, has been later extended to include numerical properties as a way to reduce the number of candidates per role (Guéhéneuc et al., 2010). DeMIMA and its extensions are available within the Ptidej tool suite.

ML has been applied to support the DPD process, mainly building classification models to characterise DPs through the inspection of code implementations. These proposals differ in terms of the role that ML plays in the detection process, the properties of the source code used for learning, and the specific algorithm applied. A first work makes use of association rules to discard classes not playing a role according to some software metrics, so ML is not directly responsible for the detection (Gueheneuc et al., 2004). Similarly, decision trees and neural networks were trained with a set of manually defined features –named predictors– to reduce the number of false positives detected after a structural analysis of the code (Ferenc et al., 2005). A collection of software metrics was used to feed a neural network, whose goal is the identification of classes potentially playing a role in the DP before addressing the detection (Uchiyama et al., 2011).

The rest of methods rely on ML techniques to make a decision about the presence of a DP in the code. Alhusain et al. (2013) train multiple artificial neural networks, one for each role, with different subsets of metrics chosen via feature selection. They also proposed a second classifier to carry out the detection after filtering some candidates as in (Uchiyama et al., 2011). In (Chihada et al., 2015), the classification model is created using support vector machines. This method uses as an input a labelled set of manually identified pattern implementations and a set of metrics associated with their roles. Then a subgraph isomorphic algorithm is executed for candidate design patterns to be extracted from the code. In MARPLE (Zanoni et al., 2015), software metrics are not taken as inputs but a set of structural and behavioural properties, such as object compositions or method delegations. Implemented as an Eclipse plugin, MARPLE makes use of several clustering and classification algorithms implemented by Weka.[1] More particularly, MARPLE is able to use different highly interpretable classifiers like decision trees. However, it does not use DP instances directly as an input. Since it executes a clustering algorithm prior to detection, it makes the resulting models not as representative of the original samples as expected and, consequently, more difficult to understand.

More recently, DPD has been treated as a multi-classification problem, where the goal is to determine which specific pattern is implemented by the input code (Dwivedi et al., 2018). Here, three different black-box classifiers — neural networks, support-vector machines and random forests — are trained using up to 67 software metrics. Convolutional neural networks and random forest have been also applied to learn from feature maps (Thaller et al., 2019). These elements are defined based on the occurrences of certain microstructures in the code.

Our proposal, GEML, belongs to the area of ML-based techniques for DPD. Compared to non-ML-based approaches, GEML does not need prior knowledge about the properties of the DP to be detected. Therefore, it can be adapted to organisational policies and different programming styles. As other ML-based methods, it automatically discovers which are the properties that best define each DP by exploring previous implementations from code repositories. As a counterpart, ML-based approaches require a labelled set of implementations, which may not be available in all contexts. Focusing on other ML-based alternatives, GEML differs from the existing approaches in the combination of software metrics and properties as input for learning. As for the algorithm, it uses G3P to produce a rule-based classifier instead of black-box detectors, such as neural networks or support vector machines. Outcomes from these kinds of models are harder to understand, which also reduces their likelihood for adoption by practitioners (Rana et al., 2014).

4. The GEML approach

Fig. 1 depicts the overall structure of GEML. Taking the approach as a black-box, the inputs required consist in the organisational repository containing patterns predicted in the past (*i.e.,* samples), and the source code for which the detection process is carried out. After the execution of GEML, the detected DPs are returned. These patterns can be incorporated to the organisational repository for future detections, so that the detection model is

[1] *Weka 3: Data Mining Software in Java*, available from `https://www.cs.waikato.ac.nz/ml/weka` (accessed June 11, 2020)

Published version available at: `https://doi.org/10.1016/j.jss.2021.110919`

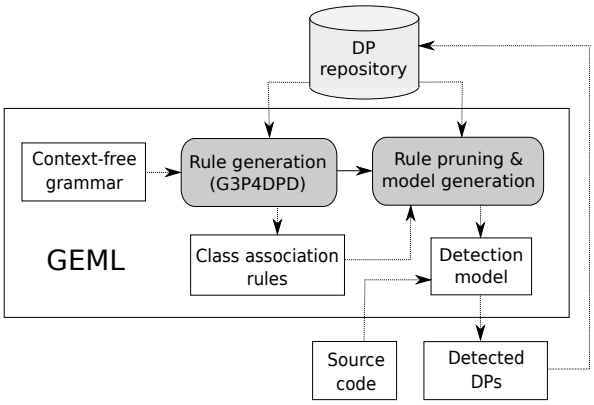


Figure 1: Two-phased model for design pattern detection

progressively adapted to the specific development culture. Going into the detail of our approach, GEML has been proposed as a two-phased model. Firstly, the set of both structural, behavioural and metric-based software properties that best describe the design pattern under analysis are learned in form of rules. With this aim, a representative set of labelled DP implementations are scrutinised from the code repository, which contains both positive and negative samples. Keeping negative samples could provide relevant information about realistic scenarios, since similarities to a real DP could lead to misclassification otherwise. In fact, a high number of negative samples benefits the robustness of the detection process, specially when the amount of positive examples in public repositories is low (Alhusain et al., 2013; Thaller et al., 2019). Every pattern instance within the repository is characterised by its constituent elements, its source code and the role mapping, *i.e.,* the correspondence between elements and roles.

The set of class association rules must be compliant with the CFG, formally defining the language syntax required to state the properties and constraints of the pattern. The G3P algorithm for DPD (G3P4DPD) has been developed to search for those rules that best characterise the implementations available in the repository. Since the resulting CARs have a descriptive nature, they could not be directly used for detection purposes. During the second step of the process, the most effective rules for the construction of the detection model will be chosen. With this aim, a pruning method is applied first to obtain the minimum set containing the most descriptive rules. Then, a strategy is followed to decide how to arrange the resulting rule set in order to build the detector, according to the precepts of different methods for associative classification (Thabtah, 2007). Both phases are explained in detail next.

4.1. G3P-based Algorithm for Rule Generation

Algorithm 1 shows the general procedure of G3P4DPD, which receives five inputs: the number of generations (*maxGen*), the population size (*popSize*), the number of individuals or CARs to be returned (*extPopSize*), the grammar (*cfg*) and the source code repository (*repo*).

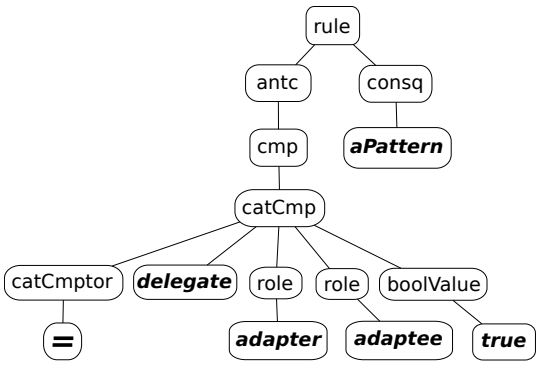
Algorithm 1: G3P4DPD

```
Input  : maxGen, popSize, extPopSize, cfg, repo
Output: extPop
pop ← generateRules(popSize, cfg)
extPop ← ∅
evaluate(pop, repo)
while generation < maxGen do
    parents ← select(pop ∪ extPop, popSize)
    offspring ← crossover(parents)
    if random() < 0.5 then
        offspring ← diversityMutator(offspring);
    else
        offspring ← dpdMutator(offspring);
    end
    evaluate(offspring, repo)
    pop ← offspring
    extPop ← update(pop ∪ extPop, extPopSize)
    generation++
end
```

G3P4DPD keeps an external archive (*extPop*) composed of the most accurate individuals according to their evaluation. They are returned as output.

Regarding its operation, G3P4DPD firstly creates a random population (*pop*) of *popSize* individuals conformant with the syntax prescribed by the grammar, *cfg*. The external archive, *extPop*, is then initialised to the empty set. Initial individuals of *pop* are evaluated by computing the fitness function, which obtains the support of the rule calculated from the repository *repo*. At this point, the algorithm iterates until the *maxGen*-th generation is reached. For each generation, the selection operator returns *popSize* individuals (parents) from the union between population *pop* and population *extPop*. Next, the crossover operator is applied with a certain probability over pairs of parents. After combining their genotypes, the obtained *offspring* are then mutated. The choice of the specific mutator is made randomly — between two possible operators — and seeks for a balance between search space exploration and the type and magnitude of the changes applied. On the one hand, *diversityMutator* looks for the improvement of population diversity by including new properties in the comparison expressions of the rule. On the other hand, *dpdMutator* is a DPD-specific mutator that aims to generate rules describing both positive and negative samples. Once these new offspring have been evaluated in terms of their support, they all replace the current population. Finally, the external archive is updated to keep those rules exceeding a certain support and confidence threshold, while ensuring that it does not exceed *extPopSize* elements, *i.e.,* rules, and they are not repeated or redundant. Next sections explain in further detail how solutions are encoded and how the aforementioned genetic operators operate.

Published version available at: https://doi.org/10.1016/j.jss.2021.110919



(a) Genotype

(= **delegate adapter adaptee true**) → (**aPattern**)

(b) Phenotype

Figure 2: Example of correspondence between genotype and phenotype for an illustrative individual

```
G = (S, ∑_N, ∑_T, P)

S = <rule>

∑_N = {<rule>, <antc>, <cmp>, <numCmp>, <numCmptor>, <numOp>,
      <catCmp>, <catCmptor>, <role>, <boolValue>, <typeOfValue>,
      <linkMethodValue>, <linkArtefactValue>,
      <ctorVisibilityValue>, <aggregationValue>,
      <adapterMethodValue>, <sameInterfaceValue>, <consq>}

∑_T = {and, ≥, ≤, >, <, =, !=, NOM, NOC, DIT, RFC, adapter,
      adaptee, target, true, false, isFinal, isSubclass,
      controlledInit, staticField, staticFlag, conglomeration,
      returned, received, createObj, delegate, sameElem, typeOf,
      linkMethod, linkArtefact,ctorVisibility, aggregation,
      adapterMethod, redirectInFamily, sameInterfaceInstance,
      sameInterfaceContainer, class, absClass, enum, intface,
      directOver, indirOver, directImpl, indirImpl, notLinked,
      directInherit, indirInherit, private, protected, package,
      public, decl, inhr, single, multi, aPattern, notAPattern}

P =
<rule>   ::= <antc> <consq>
<antc>   ::= <cmp> | and <antc> <cmp>
<cmp>    ::= <numCmp> | <catCmp>
<numCmp> ::= <numCmptor> <numOp> <role> const
<catCmp> ::= <catCmptor> isFinal <role> <boolValue>
           | <catCmptor> delegate <role> <role> <boolValue>
           | <catCmptor> typeOf <role> <typeOfValue>
           | ...
<numCmptor>    ::= ≥ | ≤ | > | <
<catCmptor>    ::= = | !=
<numOp>        ::= DIT | NOC | RFC | NOM
<role>         ::= adapter | adaptee | target
<boolValue>    ::= true | false
<typeOfValue>  ::= class | absClass | intface | enum
...
<consq>        ::= aPattern | notAPattern
```

Figure 3: Grammar used by the G3P4DPD algorithm

4.1.1. Encoding

On the one hand, the genotype of an individual is represented by a valid derivation tree of production rules, as formally defined by the CFG. On the other hand, its phenotype denotes the corresponding class association rule. As an example, Fig. 2 shows a genotype/phenotype mapping, the described rule stating that, if the element playing the role *adapter* delegates some functionality to the element *adaptee*, then this would expected to be a valid Adapter pattern.

As explained in Section 2.2, any CFG requires the definition of sets of terminal and non-terminal symbols, and the production rules to derive valid expressions from a root, non-terminal symbol. In this context, terminal symbols represent the software metrics, as well as the structural and behavioural properties that allow identifying the presence of a pattern instance, whereas non-terminal symbols define the derivable elements, including all the components needed to build comparison expressions between the aforementioned properties. Production rules determine the derivation steps that lead to the generation of detection rules, which are ultimately written in terms of terminal symbols. Notice that the CFG could be certainly customised to a specific design pattern, thus allowing a more efficient description of the most relevant properties of that pattern.

Fig. 3 shows the proposed grammar for the DPD process. Some production rules were omitted for readability. The first production rule listed in P determines that the root symbol, <*rule*>, can be derived into two other non-terminal symbols, <*antc*> and <*consq*>, representing the antecedent and the consequent, respectively. On the one hand, the antecedent can be defined as an inclusive sequence of numerical (<*numCmp*>) and categorical (<*catCmp*>) comparisons. On the other hand, <*consq*> can be derived into two possible terminals, expressing whether it is a positive detection of a design pattern instance, *aPattern*, or not, *notAPattern*. In addition, it is worth noting that this grammar has been slightly customised for the Adapter pattern, the non-terminal <*role*> being derived into *adapter*, *adaptee* and *target*. Similarly, adaptations of the grammar to other patterns could be also made straightforward by just setting the operators and roles that are more relevant to each pattern from those available, or exceptionally by implementing new operators. A more general possibility is to make the complete list of operators available to the algorithm and let it "find" which are the most relevant for the search. In any case, G3P4DPD would not be affected, since the algorithm receives the grammar definition as a parameter and it will adapt the search in consequence.

As mentioned above, each property of the DP is declared by the antecedent of the rule as a comparison, written as an expression in prefix form containing a comparator, an operator, one or more arguments, and a value. Numerical comparisons, *numCmp*, allow the representation of those properties based on software metrics. Each comparison comprises a numerical comparator ($<$, $>$, $\leq$ or $\geq$); a numerical operator that computes an object-oriented software metric from the CK (Chidamber–Kemerer) metric suite (Chidamber and Kemerer, 1994), such as NOM

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

(number of methods), NOC (number of children), DIT (depth of inheritance tree) or RFC (response for a class); an argument referring to the measured element playing a role in the pattern; and a numerical constant (*const*). As an example, the comparison "`NOC(target)<1`" indicates that the *target* class has no subclasses. Likewise, other numerical software metrics could be implemented and added to the grammar without recompiling G3P4DPD.

On the other hand, categorical comparisons involve structural and behavioural properties. Structural properties serve to express characteristics of structural elements, such as a class being abstract or final, as well as the relationships between these elements, such as an aggregation or generalisation. Behavioural properties refer to those interactions between classes or method invocations that can be analysed from static code. These comparisons are composed by a categorical comparator (= or *!=*); a categorical operator like *linkArtefact* or *delegate*; a number of arguments receiving the participating roles; and a value. As an example of a categorical comparison, "`typeOf(target)=intface`" describes that the *target* role is implemented by an interface. Table 1 shows the full list of available operators, their signatures and description. It is worth noting that categorical operators are mainly based on elemental design patterns (Smith, 2012), micro patterns (Gil and Maman, 2005) and design patterns clues (Fontana et al., 2011). They provide language elements that can be easily understood by the software engineer.

These structures can be expressed as categorical operators checking whether a code artefact satisfies a property on a true/false basis. For instance, *Abstract Interface* is an elemental design pattern checking whether a given code artefact is an interface. Usually, an artefact is a class or an interface. However, it largely depends on the specific programming language, and other artefacts could be considered like enumerations, or subtypes of artefacts, such as abstract and concrete classes. It implies that a large number of operators like *isConcrete*, *isAbstract*, *isInterface* and *isEnumeration* would be necessary to cover all types of artefacts. Besides, these operators would be strongly correlated, *e.g.,* if *isConcrete* is *true*, then the others should be *false*, what could make rules redundant and add noise to the learning process. Therefore, the CFG declares multivalued categorical operators, which are able to return different alternative terms instead. This is the case of *typeOf*, which allows analysing a source code artefact and returns a single value for all these possibilities. In this way, this operator replaces a number of redundant and unnecessarily inefficient boolean operators, while boosts the generation of more compact and readable rules.

4.1.2. Genetic Operators

There are three genetic operators to be considered in this evolutionary schema: selection, crossover and mutation. The selection operator is applied at the beginning of each generation in order to choose a set of parents for breeding. With this aim, a binary tournament selects two individuals from the union of *pop* and *extPop*, taking the best according to their fitness. This process is repeated until *popSize* parents are selected.

Regarding the crossover, this operator is applied to each pair of parents according to a given probability. The operator works by randomly selecting and swapping a single comparison from their respective genotype. Finding such a comparison requires traversing the derivation tree in preorder until a <*cmp*> symbol is reached, delimited by *and*, <*cmp*> or <*consq*>, as illustrated in Fig. 4. Then, each offspring is mutated.

Two possible variants of mutator have been considered for this problem. Firstly, the so-called *diversityMutator* aims at generating rules with novel comparisons. To this end, it selects a number of comparisons and rebuilds the individual by randomly deriving these <*cmp*> until new terminal symbols are obtained. The number of comparisons is chosen with a random roulette wheel that promotes small changes against those significantly altering the individual. As an example, Fig. 5a shows how a categorical comparison is replaced by a numerical comparison. Secondly, *dpdMutator* aims at obtaining rules describing both positive and negative samples. The rationale behind this operator is that if a comparison describes the correct implementation of a pattern, its negation should imply an incorrect result. This operator traverses the derivation tree and rewrites the logical meaning of each comparison with an inverse probability to the number of comparisons in the antecedent. With this aim, it simply switches the comparison symbol, *e.g.,* $<$ is replaced by $\geq$, as illustrated in Fig. 5b. In addition, the terminal symbol of the consequent, *i.e.,* the class label, will be also inverted with a probability of 0.5. For instance, if the comparison "`typeOf(target)=intface`" describes a recurrent property of positive samples for the Adapter pattern, "`typeOf(target)!=intface`" is likely to represent negative samples. At the end of the mutation operation, regardless of the specific variant being used, the mutated offspring is returned.

4.2. Construction of the Detection Model

After completing the first step explained above, the G3PDPD algorithm returns its external archive composed of CARs. However, they still require to be scrutinised to select strictly those that will be part of the detection model. With this aim, the database coverage method is computed on the complete set of rules. This pruning method was originally defined for CBA (Liu et al., 1998), a general approach for associative classification, and later applied by other well-known algorithms like CMAR (Li et al., 2001) or CPAR (Yin and Han, 2003).

Algorithm 2 shows the pseudocode of the pruning method, conveniently adapted to this problem. The procedure takes the archive returned by G3P4DPD, *extPop*, the repository, *repo*, and a coverage *threshold*. As an output, it returns the pruned set of rules, *ruleSet*. To do

Published version available at: `https://doi.org/10.1016/j.jss.2021.110919`

Table 1: Grammar operators to describe design pattern implementations

Numerical operators	
Signature	**Description**
NOM(r_1)	Number of methods of r_1
NOC(r_1)	Number of children directly inherited from r_1
DIT(r_1)	Depth of the inheritance tree from r_1
RFC(r_1)	Number of distinct methods and constructors potentially invoked when an object of r_1 receives a message
Categorical operators	
Signature	**Description**
isFinal(r_1)	*true* if r_1 cannot be extended; *false* otherwise
isSubclass(r_1)	*true* if r_1 is a subclass; *false* otherwise
controlledInit(r_1)	*true* if r_1 instantiates itself within an *if* or *while* block; *false* otherwise
controlledExcept(r_1)	*true* if r_1 uses exceptions and an static flag to control its instantiation; false otherwise
conglomeration(r_1)	*true* if 2 or more methods of r_1 are invoked from another method from r_1; *false* otherwise
returns(r_1,r_2)	*true* if a value of type r_2 is returned from a method of r_1; *false* otherwise
receives(r_1,r_2)	*true* if a method of r_2 receives a value of type r_1 as argument; *false* otherwise
createObj(r_1,r_2)	*true* if r_1 instantiates r_2; *false* otherwise
delegates(r_1,r_2)	*true* if a method of r_1 invokes a method of r_2; *false* otherwise
sameElem(r_1,r_2)	*true* if r_1 and r_2 are coded by the same artefact; *false* otherwise
typeOf(r_1)	Returns the type of the artefact implementing r_1 (*absClass*, *intface*, *enum*, *class*)
linkMethod(r_1,r_2)	Indicates if a method of r_1 directly or indirectly overrides (*directOver*, *indirOver*), implements (*directImpl*, *indirImpl*) or is *notLinked* to a method of r_2
linkArtefact(r_1,r_2)	Returns the sort of link between the artefacts playing r_1 and r_2 (*directInherit*, *indirInherit*, *directImpl*, *indirImpl*, *notLinked*)
ctorVisibility(r_1)	Returns the visibility of the less restrictive constructor of r_1 (*private*, *protected*, *package*, *public*)
aggregation(r_1,r_2)	Returns information about an attribute of r_2 declared in r_1 in terms of its visibility and instantiability
adapterMethod(r_1,r_2,r_3)	Returns if a declared (*decl*) or inherited (*inhr*) method of r_1, implemented from r_3, delegates in a method of r_2; *notLinked* otherwise
redirectInFamily(r_1)	Returns if a declared method of r_1 is delegated in a class or interface being extended or implemented by r_1, once (*single*) or multiple times (*multi*); *notLinked* otherwise
sameInterfaceInstance(r_1,r_2)	Returns if $r2$ is a class or interface being extended or implemented by r_1, which has one (*single*) or multiple (*multi*) fields of a class or interface being extended or implemented by r_2; *notLinked* otherwise
sameInterfaceContainer(r_1,r_2)	*true* if $r2$ is a class or interface being extended or implemented by r_1, which defines a collection of a class or interface being extended or implemented by r_2; *false* otherwise

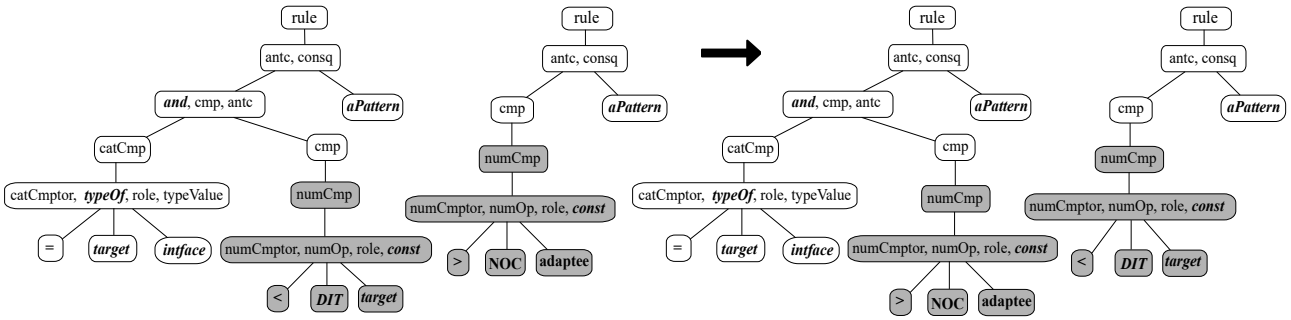
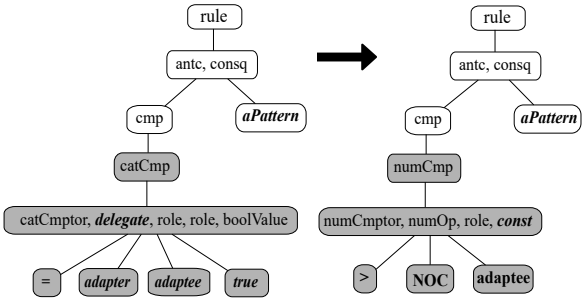


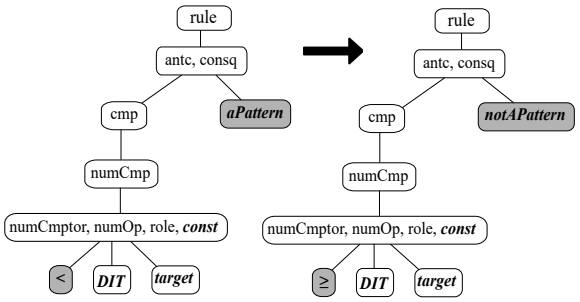
Figure 4: Example of the crossover operator

this, rules are sorted according to their confidence, support and size, *i.e.,* number of comparisons, respectively. More precisely, given two rules R_1 and R_2, it is said that R_1 *precedes* R_2 iff any of these conditions is satisfied: (*a*) $\text{conf}(R_1) > \text{conf}(R_2)$; (*b*) $\text{conf}(R_1) = \text{conf}(R_2)$ and $\text{supp}(R_1) > \text{supp}(R_2)$; or (*c*) $\text{conf}(R_1) = \text{conf}(R_2)$, $\text{supp}(R_1) = \text{supp}(R_2)$ and $\#comparison(R_1) < \#comparison(R_2)$. Then, all repository samples are scanned for each rule searching for those matching the antecedent. When positive, they are conveniently added to the list *covSamples*. In addition, if the rule correctly classifies at least one of these samples, *i.e.,* the flag *marked* is true, then it will

Published version available at: https://doi.org/10.1016/j.jss.2021.110919



(a) Operator *diversityMutator*



(b) Operator *dpdMutator*

Figure 5: Examples of the mutation operators

Algorithm 2: Database coverage

```
Input  : extPop, repo, threshold
Output: ruleSet
ruleSet ← ∅
sort(extPop)
foreach rule in extPop do
    marked ← false
    covSamples ← ∅
    foreach sample in repo do
        if match(rule, sample) then
            covSamples ← covSamples ∪ sample
            if rule correctly classifies sample then
                marked ← true
            end
        end
    end
    if marked then
        ruleSet ← ruleSet ∪ rule
        increaseCoverage(repo, covSamples)
        removeCoveredSamples(repo, threshold)
    end
    if empty(repo) then
        break
    end
end
```

be added to *ruleSet* and the counter of covered samples in *repo* will be increased. In case a sample is covered by a number of rules greater than or equal to the *threshold*, it will be removed from *repo* so that this sample does not need to be checked again. Finally, if the rule does not contribute to increase the current detection capability, it will be discarded. This happens when the rule detects the same samples as other rules with higher confidence and support. This process is repeated until all samples in *repo* have been covered or until there are no more rules left.

After the pruning procedure, the selected rules need to be arranged in order to constitute the classifier. Here, the four most commonly referenced strategies from the field of associative classification have been considered for comparison (Thabtah, 2007): maximum likelihood (MAXL); dominant factor multiple label (DFML); DFML, as defined by CMAR (DFML$_{\chi^2}$); and DFML, as determined by CPAR (DFML$_{Lap}$). They all operate differently when a new, unknown sample is received. For instance, the MAXL method selects the highest ranked rule covering the new sample, its consequent implying the prediction. This approach has been criticised by some authors since a single rule becomes responsible for the decision of the classifier (Li et al., 2001). In contrast, DFML selects all rules covering the incoming sample and distributes them according to their consequent (Hadi et al., 2016). The partition containing more rules determines the class. This method has been adapted in different proposals by changing how the most representative partition is obtained. A well-known variant is DFML$_{\chi^2}$, where the weighted χ^2 is computed for each partition, the set of rules with the highest value being returned. This measure analyses the strength of the rules based on their support and class frequency. Similarly, DFML$_{Lap}$ calculates the *Laplace* accuracy to estimate the expected accuracy for each rule and selects the partition with its highest average. It should be noted that DFML$_{Lap}$ considers only the best k rules within each partition, k being a prefixed parameter. Then, after sorting the rules, the classifier is ready to receive new code, apply the rules and predict the presence of a DP implementation.

5. Experimental Settings

We first pose the research questions that set out the objectives to be validated in this section. Then we present the data repositories and explain the methodology followed in the experiments. The last section details the experimental framework.

5.1. Research Questions

The conducted experiments aim at responding the following research questions (RQs):

- *RQ1. What configuration best informs our technique for each DP?* The combination of parameters for the G3P-based machine learning algorithm and the pruning method might lead to vast number of results. It would be convenient to objectively provide the best parameter setup for each DP.

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

- *RQ2. For the software engineer, which are the most significant design elements and metrics to be considered for the detection of a given pattern?* The combination of numerical (metrics) and categorical (behavioural and structural properties) attributes offers the engineer the possibility of using a wide range of microstructures and operators at his/her convenience. In this way, after analysing which operators are most frequently used to describe each pattern, the experimentation can be replicated by adjusting them in order to find out whether the results maintain the detection performance.

- *RQ3. Does GEML provide as much effectiveness as other ML-based DPD techniques?* It would be convenient to study if, taking a single configuration for all DPs, our model is still competitive against a similar ML-based approach. Reaching a general-purpose configuration would relieve the software engineer of the need to set up the model.

- *RQ4. How does training conditions influence GEML?* As any other ML-based approach, GEML requires the availability of DP examples to learn from. Therefore, it is important to study how GEML behaves when few examples are provided. This would allow the software engineer to confirm the potential of the technique under different execution scenarios.

- *RQ5. Does GEML provide as much effectiveness as other non-ML-based DPD techniques?* Similarly to RQ3, it would be interesting to analyse the benefits that GEML brings compared to other non-ML-based DPD tools currently available to the software engineer, thus ensuring that it also contributes to advance in the state of the practice.

5.2. Experimental Data Sources

The experimentation considers a total of 15 design patterns, covering the three different categories defined by (Gamma et al., 1995): creational (Abstract factory, Factory method and Singleton), behavioural (Command, Iterator, Observer, State, Strategy, Template Method and Visitor) and structural (Adapter, Bridge, Composite, Decorator and Proxy). These patterns provide different levels of complexity, as they differ in the number of roles and may present elements playing multiple roles.

Samples of these design patterns have been extracted from public repositories, but other institutional data sources could be valid too. More specifically, we conduct our experiments with implementations from two repositories, namely DPB and P-Mart, each with a different purpose. Firstly, we consider DPB (Fontana et al., 2012) because it provides the highest number of samples from publicly available repositories. Moreover, DPB was created by the authors of MARPLE to validate their approach, what ensures a fair comparison against this particular proposal, and contains both positive and negative samples, a requirement when experimenting with ML-based methods. DPB collects samples from nine industrial Java projects, plus one more project adopted from the literature. Table 2 lists the number of artefacts, methods, attributes and lines of code (LOC) of each software project contained by DPB. A limitation of this repository is that it only supports five design patterns (see Table 3, which includes the total number of samples [S], divided into positive [+] and negative [-]).

In order to compare GEML against other non-ML-based proposals, we rely on P-Mart (Guéhéneuc, 2007), a peer-validated repository frequently used in other DPD studies. P-Mart includes samples of 20 design patterns distributed among all projects listed in Table 2, with the exception of the DPExample project. Table 4 summarises the samples per project identified within P-Mart, as well as the number of existing DPs from those defined by Gamma. As can be observed, JHotDraw is the project containing the greatest variety of DPs (11), making it appropriate for its use as a test project when comparing GEML with other DPD studies, as will be explained later in Section 5.3.

The use of P-Mart allows GEML to be set against other DPD methods reporting results for JHotDraw, but this project limits the number of DPs available for comparison. Notice that P-Mart was not originally designed to specifically experiment with ML-based approaches due to the reduced number of positive samples, so it has to be enlarged to be suitable for training (Thaller et al., 2019). To carry out a more complete, accurate and practicable analysis, it is important to provide GEML with a more extensive collection of DPs for training than those available in P-Mart. Therefore, we have built a training repository in order to validate the applicability of GEML on an even greater number of design patterns. This is a common practice in DPD studies, especially in those presenting ML-based proposals (Ferenc et al., 2005; Alhusain et al., 2013; Thaller et al., 2019). Therefore, we have complemented the positive samples available in P-Mart with other instances manually validated from those recovered by DPD tools (Ferenc et al., 2005; Lucia et al., 2018). In particular, we include those pieces of code for which both SSA and Ptidej reported the presence of a DP implementation with all its roles. In the case of Ptidej, this information is complete. In contrast, some samples of P-Mart are not fully labelled, which has caused us to double-check the repository to maintain only those instances that describe all their roles. As for the negative samples, a common approach is to generate random candidates from the code and select those that are more similar to positive samples according to some heuristics (Thaller et al., 2019). We follow a similar procedure, so that a maximum of three negative samples per positive sample have been generated.[2] The final list of design patterns available in the repository is

[2]Details of the procedure and the resulting training repository are available from the additional material (see page 23).

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

Table 2: Properties of the Java projects used for experimentation

Project	Types	Methods	Attributes	LOC
DPExample	1749	4710	1786	32313
QuickUML 2001	230	1082	421	9233
Lexi v0.1.1 alpha	100	677	229	7101
JRefactory v2.6.24	578	4883	902	79732
Netbeans v1.0.x	6278	28568	7611	317542
JUnit v3.7	104	648	138	4956
JHotDraw v5.1	174	1316	331	8876
MapperXML v1.9.7	257	2120	691	14928
Nutch v0.4	335	1854	1309	23579
PMD v1.8	519	3665	1463	41554

Table 3: Samples per design pattern in each experiment

Design pattern	Experiment #1			Experiment #2		
	+	-	S	+	-	S
Adapter	618	603	1221	65	180	245
Fact. Method	562	482	1044	3	9	12
Decorator	93	154	247	2	6	8
Singleton	58	96	154	55	165	220
Composite	30	98	128	6	18	24
State	-	-	-	147	412	559
Templ. Method	-	-	-	50	147	197
Proxy	-	-	-	19	57	76
Observer	-	-	-	8	14	22
Strategy	-	-	-	7	21	28
Abs. Factory	-	-	-	6	18	24
Command	-	-	-	5	15	20
Visitor	-	-	-	3	9	12
Iterator	-	-	-	3	9	12
Bridge	-	-	-	2	6	8

Table 4: Characteristics of the projects available in P-Mart and DPExample

Project	No. DPs	No. positive samples
DPExample	19	174
QuickUML 2001	6	7
Lexi v0.1.1 alpha	3	5
JRefactory v2.6.24	6	26
Netbeans v1.0.x	4	26
JUnit v3.7	5	8
JHotDraw v5.1	11	21
MapperXML v1.9.7	9	15
Nutch v0.4	8	15
PMD v1.8	9	14

shown in Table 3, together with the number of training samples. After preliminary experiments, the number of positive samples for the remaining seven DPs available in P-Mart has proven to be insufficient for training.

5.3. Description of Experiments

Three experiments have been planned in order to find an answer to these RQs:

- *Experiment #1.* In this experiment, we validate the detection capability of GEML under laboratory settings. More specifically, we study how the parameter tuning influences our proposal and analyse different configurations with respect to MARPLE (Zanoni et al., 2015), another method based on machine learning.
- Experiment #2. In this experiment we compare GEML against other recent non-ML-based DPD proposals: DePATOS (Yu et al., 2018), MLDA (Al-Obeidallah et al., 2018) and SparT (Xiong and Li, 2019). With this aim, for comparison purposes, we provide results on a P-Mart project, JHotDraw.
- *Experiment #3.* This experiment focuses on more qualitative aspects, evaluating GEML in a more practical setting. Here, GEML is tested on a project, DPExample, taken from a different repository, DPB, than the one used for training. In this case, we compare our proposal with two non-ML-based DPD tools, SSA and Ptidej, which were run using the testing project as input. We choose these tools because they are the most frequently used for comparative purposes in DPD studies (selected in 88% and 35% of the works, respectively). In our preliminary analysis, we found other eight tools, but they were unavailable for download or failed in their installation or execution.

To cope with the intrinsic randomness of evolutionary algorithms, 30 independent runs with different random seeds are executed. For Experiment #1, the possible bias due to the training partitions of the input repository (DPB) is solved by carrying out a stratified 10-fold cross validation in every execution. In Experiment #2, we set the conditions that allow the comparison with other DPD approaches on a widely studied project, JHotDraw. All the DPD under analysis provide access to the recovered instances in this project, allowing a fair comparison by inspecting their level of agreement with respect to P-Mart. Also, JHotDraw is the P-Mart project with a greater variety of DP implementations. The rest of projects within P-Mart are used as the training set. Here we only consider as truly DP samples those labelled as positive on P-Mart. In Experiment #3 we want to simulate the process in which an engineer uses past projects to train the DPD model over all the possible DPs, and then apply the detection model on a new project. Since none of the nine P-Mart projects contains samples of all the DPs, we used them together for training, and the detection model is tested on DPExample. Since SSA and Ptidej do not require any training phase, they are directly executed using DPExample as input. True positive samples of DPExample have been manually revised.

Results are then reported in terms of the following commonly used classification measures: *accuracy*, which indicates the percentage of instances correctly classified; *recall*, which corresponds to the amount of DPs retrieved over the total number of samples within the repository; *precision*,

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

which measures how many DPs are positively detected as positive; *specificity*, which indicates the proportion of negative samples correctly identified as negative; and F_1 *score*, which calculates the harmonic mean between precision and recall. Once these performance metrics have been computed, the relevance of the results (Arcuri and Briand, 2014) needs to be statistically validated. Firstly, the Wilcoxon test allows performing pairwise comparisons, where the null hypothesis, H_0, hypothesises that both algorithms – or configurations – perform equally well. The distributions to be compared represent the results of 30 executions of the evolutionary algorithm for a given DP and performance measure. As multiple configurations are tested, p-values are conveniently adjusted by using the Holm's method. For those pairwise comparisons reporting significant differences, the Cliff's delta test is carried out to measure the effect size. A 95% confidence level is considered for both tests.

5.4. Experimental Framework

An implementation of GEML is provided in Java using JCLEC (Ventura et al., 2008), which includes functionalities and data structures to implement evolutionary algorithms. The VF2 algorithm (Cordella et al., 2004), available in the VFLib graph matching library,[3] has been used for implementing the generation of candidates in Experiment #2. Three additional libraries have been used: ckjm,[4] an implementation of the CK metric suite; Java Parser,[5] a source code parser used to extract information and properties from code; and Javassist,[6] a similar library that takes bytecode as input. They both are used to implement the grammar operators during rule evaluation, although bytecode-based operators are preferred because they allow a faster computation of the properties.

Table 5 lists the parameter setup and its variations for the detection model, including the G3P4DPD algorithm and the pruning procedure. The population size, the number of generations and the crossover probability have been set after preliminary experiments. The maximum number of derivations determines the limit of the genotype size, *i.e.,* how long a rule can be. This value has been set to 25, as larger values would hardly generate individuals with admissible support. In addition, the size of the external archive has not been restricted to avoid discarding interesting rules and to test the effectiveness of the pruning method. The rest of parameters will be set according to the conclusions extracted from the parameter study in Section 6.1, which serves to determine their influence on the detection process. As for this analysis, support, confidence and coverage are configured according to the most frequently applied values within the AC field (Liu et al., 1998; Li et al., 2001).

Table 5: Parameter setup of the DPD model

Parameter	Value
Population size	100
Number of generations	150
Crossover probability	0.8
Maximum number of derivations	25
Coverage threshold	1, 2, 3, 4
Support threshold	0.01, 0.05, 0.1
Confidence threshold	0.5, 0.6, 0.7

6. Experiment 1: Validation of the Detection Model

Experiment #1 is explained in this section. The internal elements of the detection model are analysed by means of an extensive parameter study (RQ1). Then, we also focus on the selection of the grammar operators, revealing the most recurrent microstructures and design elements to describe each DP (RQ2). Finally, the results are discussed with special emphasis on the effectiveness of the proposal (RQ3).

6.1. Parameter Study

In response to RQ1, we carry out a parameter study focused on the three influential aspects required for the most fitting parametrisation of GEML: the coverage threshold, which is a key parameter for rule pruning; support and confidence thresholds, which effect the production of high-quality rules during the evolutionary search; and the strategy selection, which determines how the pruned rules are arranged to form the DPD model (see the Additional Material and Appendix for complete results).

6.1.1. Coverage Threshold

The coverage threshold determines how many rules a given sample should satisfy to be considered as covered by the detection model. The lower the threshold, the lower the number of resulting rules. Following the recommendations of the AC literature (Liu et al., 1998; Li et al., 2001), we have selected four possible values, from 1 to 4. Default values for support and confidence have been set to 0.01 and 0.5. In addition, given that a classification strategy is required, MAXL has been chosen as the baseline procedure, since it is the simplest, only considering one rule for the classification. Again, the five classification measures are computed for each configuration, with special emphasis in F_1, which reflects the effect of both precision and recall. After 30 executions, the best average values for F_1 were obtained when the coverage threshold is equal to 1. As for the statistical analysis, the Wilcoxon test does not report significant differences in the case of

[3] *VFLib*, available from `https://mivia.unisa.it/vflib/` (accessed June 20, 2020)

[4] *Chidamber and Kemerer Java Metrics (ckjm)*, available from `https://www.spinellis.gr/sw/ckjm` (accessed June 20, 2020)

[5] *JavaParser for processing Java code* available from `https://javaparser.org/` (accessed June 20, 2020)

[6] *Javassist: Java bytecode engineering toolkit* available from `http://www.javassist.org/` (accessed June 20, 2020)

Published version available at: `https://doi.org/10.1016/j.jss.2021.110919`

the Adapter. In contrast, differences were found for the rest of patterns, especially when compared with largest values of the threshold. Considering these differences and the fact that lower coverage thresholds help reducing the size of the rule set, the threshold is set to 1 for all the patterns.

6.1.2. Support, Confidence and Classification Strategy

The update mechanism of the archive is controlled by the support and confidence thresholds, looking for only preserving high-quality rules. In response to RQ1, we need to determine the parameters that return rules with the highest quality for each DP. With this aim, all the combinations of support and confidence thresholds (see Table 5), jointly with different classification strategies, have been analysed. The outcomes from these combinations can be found in the Appendix for each design pattern separately. Table 6 summarises the findings, showing the best combination of parameters with respect to F_1.

For the sake of clarity, results are mainly analysed based on the values obtained for F_1. In addition, notice that accuracy is not a fully reliable measure here, as long as the DPB repository is highly imbalanced. With respect to the classification performance of the different configurations, there is not a big difference for the Singleton, Adapter and Factory Method patterns. The same does not apply for the Decorator and Composite, however, for which differences of more than 20% are reported. These differences mainly occur between those configurations using different classification strategies, the support and confidence thresholds having less effect.

Regarding the classification strategy, it is worth noting that the same set of CARs has been used as input for all the strategies. In the case of DFML$_{Lap}$, k is set to 5, as suggested by its authors. Firstly, the results reveal that using only one rule for detection is not the best alternative, since MAXL is never able to return the best value for any performance measure. In fact, this strategy reaches significantly worse results for the Decorator and Composite patterns. Similarly, DFML$_{Lap}$ obtains low detection performance for these patterns, even worse than MAXL. Note that this strategy only takes into account a reduced number of rules for detection, which is given by the value of k. Thus, it could be argued that using a limited number of rules is not the best alternative for the DPD problem. Secondly, considering the strategies DFML and DFML$_{\chi^2}$, the latter stands out as the procedure that best exploits the detection capability of GEML. More specifically, the best values of F_1 are reached when using this strategy. Focusing on the negative samples, other strategies could obtain better specificity results but at the expense of lower recall values, meaning that less DP instances would be recovered. Therefore, only configurations using DFML$_{\chi^2}$ are considered when analysing how the support and confidence thresholds influence the detection model.

As for the support and confidence thresholds, the best performance is mostly reached for those configurations with a higher confidence threshold (0.7), the Composite being the only exception. We speculate that these rules are expected to provide more certainty in the detection. Nevertheless, notice that lower values are commonly used in the AC literature. Regarding the support, the Singleton and Adapter obtain the best results with the lowest support value (0.01), whilst 0.05 is the best value for the Factory Method, Decorator and Composite patterns. However, there is no best configuration with a support value of 0.1. Since this measure is related to the proportion of samples satisfying a rule, low values are needed to include those rules that are able to describe less common pattern implementations within the repository. Therefore, lower support values stand out as the best alternative, as suggested by the AC literature.

Finally, we summarise the main findings that give answer to RQ1:

- Low support values are preferred for all design patterns, 0.01 showing better results for Singleton and Adapter. For Factory Method, Decorator and Composite, a support equal to 0.05 is recommended.
- A confidence threshold equal to 0.7 is a good general choice, as only Composite obtains better results when setting another value (0.6).
- For all design patterns, DFML$_{\chi^2}$ is the pruning strategy providing best performance. This strategy is especially beneficial for the Decorator and Composite patterns.

6.2. Selection of Grammar Operators

The CFG determines the type of expressions (see Section 4.1.1) to appear within the rules describing DP implementations. As shown in Section 6.1, all these operators could be applied without further parametrisation to capture any type of design pattern under analysis, making the practical use of this detection model easier to the software engineer. Nevertheless, engineers could discard from the grammar those operators referred to microstructures that, in their opinion, are not descriptive of the pattern to be detected, or even are modified to comply with their organisational practices. It seems natural to think that the selection of a number of representative operators would produce rules formed from a limited set of pre-selected elements. In addition, it could reduce the search space and, consequently, the time required to find the best rules. Therefore, in response to RQ2, it is interesting to analyse to what extent the selection of operators influence the search process and which operators provide a better detection capability for each design pattern. With this aim, we have counted the number of occurrences in the pruned set of rules for the five design patterns after 30 executions. The results are depicted in Fig. 6 as box-plots, where the frequency of appearance has been normalised to the range [0, 1].

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

Table 6: Best classification performance in terms of F_1 for each design pattern

	Strategy	Supp-Conf	Accuracy	Precision	Recall	Specificity	F_1
Singleton	$DFML_{\chi^2}$	(0.01) - (0.7)	0.9561 ± 0.0136	0.9460 ± 0.0147	0.9461 ± 0.0298	0.9621 ± 0.0118	0.9411 ± 0.0202
Adapter	$DFML_{\chi^2}$	(0.01) - (0.7)	0.8688 ± 0.0022	0.8430 ± 0.0028	0.9121 ± 0.0032	0.8244 ± 0.0038	0.8757 ± 0.0020
Factory Method	$DFML_{\chi^2}$	(0.05) - (0.7)	0.8304 ± 0.0082	0.8113 ± 0.0081	0.8965 ± 0.0158	0.7533 ± 0.0145	0.8503 ± 0.0081
Decorator	$DFML_{\chi^2}$	(0.05) - (0.7)	0.8229 ± 0.0179	0.8043 ± 0.0269	0.7251 ± 0.0393	0.8824 ± 0.0171	0.7501 ± 0.0302
Composite	$DFML_{\chi^2}$	(0.05) - (0.6)	0.8859 ± 0.0181	0.7567 ± 0.0408	0.8900 ± 0.0700	0.8845 ± 0.0243	0.7885 ± 0.0425

For the Singleton pattern, it can be observed that those operators related to the class instantiation, such as *ctorVisibility* or *controlledExcep* have a strong presence, whereas those requiring more than one role as input are hardly used. As for the Adapter, *adapterMethod*, which is a pattern-specific operator, appears in most of rules, whereas *sameElem* or *sameInterfaceContainer* are rarely selected. As expected, those operators related to object creation like *createObj* and *returns* commonly appear in the rules describing the samples of the Factory Method. Finally, *redirectInFamily*, *sameInterfaceInstance* and *sameInterfaceContainer* are recurrent operators for the Decorator and Composite patterns, as they are related to delegations between classes belonging to the same inheritance tree. The ability of G3P4DPD to mine negative rules is the reason why almost every operator appears in the final rule set. Notice that using a reduced set of operators would require a smaller number of generations, *i.e.,* 100, to complete the search. Besides, including (potentially) dispensable operators within a rule could affect its readability too. Therefore, those operators whose mean frequency is less than or equal to 0.05 have been omitted from the experiments in order to find out how this selection would affect the detection performance. This value, depicted as dashed lines in Fig. 6, has been set to all DPs after preliminary experimentation.

The experimentation has been carried out analogously to the previous experiments without selection of grammar operators. For brevity, only the best results will be shown (for a complete list of results, see Additional Material). Table 7 lists the best results obtained for each DP considering the reduced set of grammar operators. Figures in bold typeface represent those values that improve the analogous results — i.e. same support, confidence and strategy — obtained without reducing the set of operators (see Table 6). As can be observed, $DFML_{\chi^2}$ is still the classification strategy obtaining the best results for every pattern. Regarding the confidence threshold, best values are mostly reached for configurations with a high value (0.7), the Composite being the only requiring a lower value (0.5). Again, a high support value (0.1) is never recommended. In general, note that the resulting values remain robust and there are not notorious differences with previous experiments including all operators. As for F_1, in absolute terms, the selection seemingly benefits the results obtained for Singleton and Composite. In the case of the Adapter, values with or without selecting operators reflect practically a tie. We argue that this consistency in the results favours the engineers feeling able to adapt the selection of grammar operators either to the needs of their organisational repositories or to reduce the search space without significantly affecting the detection performance. Pairwise comparisons have been performed between analogous configurations having and having not reduced the set of grammar operators. This statistical analysis reflects that there are no significant differences, the Factory Method being the only exception, as it slightly improves when all operators are used.

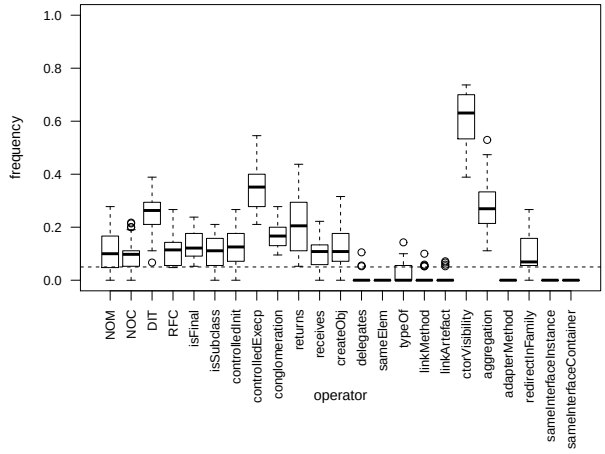
To sum up, the following insights about the impact of the grammar operators can be extracted (RQ2):

- GEML is able to automatically determine the type of operators more relevant to each category of design pattern. Those related to visibility and instantiation allow detecting creational patterns (Singleton and Factory method), whereas operators focused on delegation structures are highly effective to recover structural and behavioural patterns (Adapter, Decorator and Composite).
- The general performance of GEML does not significantly decrease when the set of operators is reduced, Factory method being the only exception.

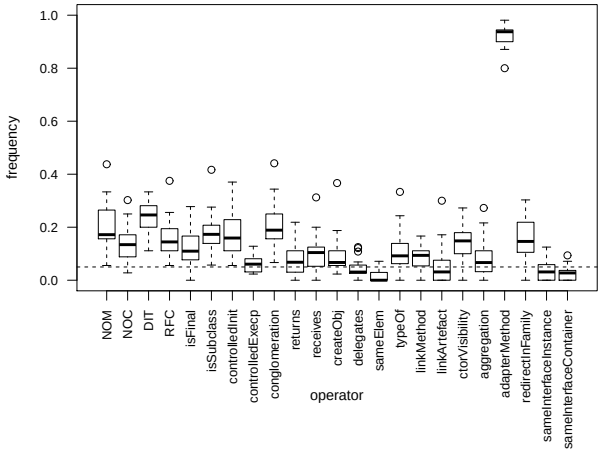
6.3. Performance Comparison and Discussion

Performance comparison. Section 6.2 showed how robust GEML behaves when one specific part of the configuration, *i.e.,* the grammar operators representing design microstructures, is refined. Nevertheless, even though it is likely that software engineers could make design-based decisions like selecting these microstructures, it still seems unrealistic to think that they would have the skills required to set up the rest of parameters and adjust the model to their needs. In these sense, for the sake of practicality, we have considered the use of a common model configuration for all design patterns taking the results listed in Section 6.1. As previously observed, $DFML_{\chi^2}$ clearly dominates the rest of classification strategies. As for the support and confidence thresholds, S(0.01)-C(0.7) reaches the best values for the Singleton and Adapter, whereas S(0.05)-C(0.7) is the most appropriate for the Factory Method and Decorator. Finally, S(0.05)-C(0.6) was the best configuration for the Composite pattern. As lower values of support are preferred in order to find rare DP implementations, S(0.01) is seemingly a comprehensive choice. Similarly, C(0.7) is selected as the confidence threshold since it reaches the best values for all the cases, except for the Composite, for which also gets competitive

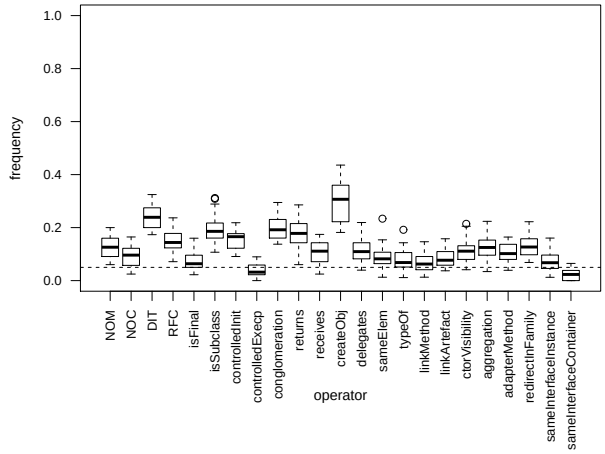
Published version available at: https://doi.org/10.1016/j.jss.2021.110919



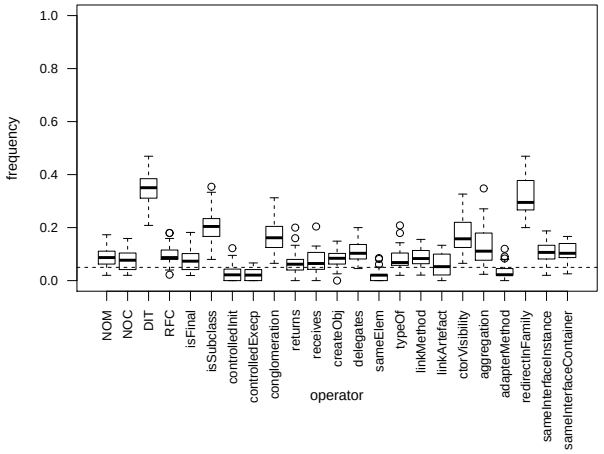
(a) Singleton



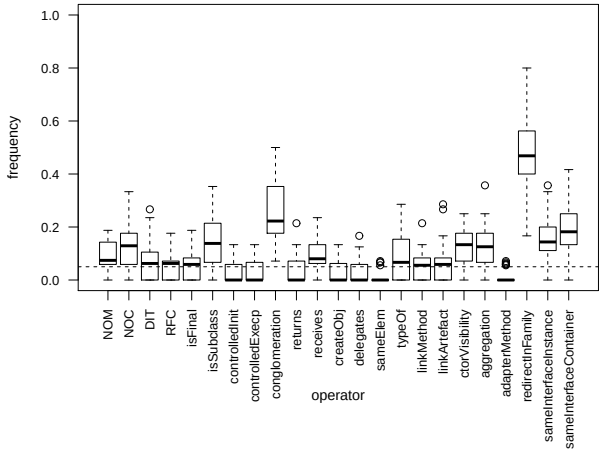
(b) Adapter



(c) Factory Method



(d) Decorator



(e) Composite

Figure 6: Frequency of appearance of grammar operators in the resulting rules

performance values. The results obtained for this configuration for the five patterns are also shown in Table 8, which compiles the results obtained during the parameter study. More precisely, it shows the accuracy and F_1 returned by the best configuration —in terms of F_1— found for each design pattern, as well as the default common configuration discussed above.

In response to RQ3, the detection performance of GEML is compared against MARPLE, a ML-based tool experimented under the same experimental methodology.

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

Table 7: Best results for the operator study

	Strategy	Supp-Conf	Accuracy	Precision	Recall	Specificity	F_1
Singleton	$DFML_{\chi^2}$	(0.01) - (0.7)	**0.9596 ± 0.0115**	**0.9492 ± 0.0180**	**0.9508 ± 0.0251**	**0.9649 ± 0.0128**	**0.9452 ± 0.0173**
Adapter	$DFML_{\chi^2}$	(0.01) - (0.7)	0.8686 ± 0.0017	0.8428 ± 0.0028	**0.9125 ± 0.0026**	0.8236 ± 0.0033	0.8756 ± 0.0016
Factory Method	$DFML_{\chi^2}$	(0.01) - (0.7)	0.8230 ± 0.0099	0.8097 ± 0.0082	0.8816 ± 0.0209	0.7545 ± 0.0158	0.8423 ± 0.0107
Decorator	$DFML_{\chi^2}$	(0.05) - (0.7)	**0.8236 ± 0.0129**	**0.8054 ± 0.0271**	0.7196 ± 0.0246	0.8868 ± 0.0175	0.7490 ± 0.0193
Composite	$DFML_{\chi^2}$	(0.05) - (0.5)	**0.8840 ± 0.0187**	**0.7438 ± 0.0465**	**0.9089 ± 0.0564**	**0.8763 ± 0.0271**	**0.7937 ± 0.0345**

Table 8: Comparison results for Experiment #1

	Best configuration				**General-purpose configuration**			
	GEML		**MARPLE**		**GEML**		**MARPLE**	
	Accuracy	F_1	Accuracy	F_1	Accuracy	F_1	Accuracy	F_1
Singleton	**0.9561**	**0.9411**	0.88	0.91	**0.9561**	**0.9411**	0.93	0.90
Adapter	**0.8688**	**0.8757**	0.86	0.85	**0.8688**	**0.8757**	0.85	0.84
Factory Method	**0.8304**	**0.8503**	0.82	0.83	**0.8308**	**0.8489**	0.82	0.83
Decorator	0.8229	0.7501	0.82	**0.77**	0.8188	0.7448	0.82	**0.77**
Composite	**0.8859**	**0.7885**	0.77	0.56	**0.8888**	**0.7568**	0.75	0.45

More specifically, we compare the best results obtained by MARPLE for each pattern, as originally reported by the authors (Zanoni et al., 2015). The general-purpose configuration for MARPLE corresponds to a Random Forest classifier with k-Means as preprocessor, as this combination provided the best results for three out of the five patterns. Comparative results can be found in Table 8, considering the best and the general-purpose configuration for both techniques. As can be observed, considering the five DPs in this experiment, GEML outperforms MARPLE for the Singleton, Adapter and Factory Method, with percentage of improvements equal to 4.57%, 4.25% and 2.28%, respectively, when comparing general-purpose configurations. In contrast, MARPLE achieves an increase of 2.65% in F_1 for the Decorator. However, GEML demonstrates a significant advantage in the case of the Composite pattern, obtaining an increase of 35.14% in F_1 when comparing the results of our general-purpose configuration even against the best results of MARPLE, and 68.18% if we compare their best configurations. Again, it is worth remarking that GEML reaches more stable values along the different design patterns, in contrast to MARPLE.

GEML provides better detection performance for the Singleton pattern, whose structure exhibits one single role. In contrast, the worst results are returned for the Decorator and Composite. This is in line with the conclusions drawn by other DPD models (Uchiyama et al., 2011; Chihada et al., 2015; Zanoni et al., 2015), where the detection capacity decreases as the pattern complexity increases.

From the comparison results, the following findings related to RQ3 can be extracted:

- GEML outperforms MARPLE for four out the five patterns under comparison, achieving up to 68.18% of improvement in terms of F1 for the Composite pattern.
- GEML provides more stable results than MARPLE, with more than 81% of accuracy and 74% of F1 for all patterns, using either its general-purpose or best configuration.
- Like other ML proposals, GEML shows a better detection performance for less complex patterns.

Discussion of design characteristics. GEML presents some characteristics in its design that makes it an appealing alternative beyond its good performance. We next discuss their implications from a more practical perspective. The use of a CFG and a large, diverse set of eligible design properties in terms of grammar operators facilitates the customisation of the detection process to the engineers' needs. In contrast to other proposals, GEML allows both numerical and categorical properties, meaning the simultaneous use of metrics and design microstructures. In this way the software engineer can explicitly influence the design and developmental aspects that will guide the search. Furthermore, the resulting model would be able to capture the most significant properties in each moment, adapting the model to changes in the repository. Promoted by the use of ML techniques, new detected samples — both positive and negative — by the resulting models could also serve as an input for future detections. Thus, GEML would allow the progressive adjustment of the prediction to the organisational development culture, what is usually a dynamic element in companies. Finally, in comparison to other black-box proposals (Chihada et al., 2015; Dwivedi et al., 2018; Thaller et al., 2019), GEML uses a rule-based model, which is more comprehensible for practitioners (Kotsiantis et al., 2006).

7. Experiment 2: Comparison Against DPD Methods Using P-Mart

This section presents the results and analysis of Experiment #2, in which GEML is compared to other DPD studies considering a project available in P-Mart, JHotDraw, as the testing project. We provide results from 10 out of the

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

Table 9: Comparison results for JHotDraw project.

	Ground	GEML					DePATOS					SparT					MLDA				
	truth	P	TP	Pr	Re	F_1	P	TP	Pr	Re	F_1	P	TP	Pr	Re	F_1	P	TP	Pr	Re	F_1
Adapter	1	1	1	1.00	1.00	1.00	38	1	0.03	1.00	0.05	12	1	0.08	1.00	0.15	19	1	0.05	1.00	0.10
Command	1	11	1	0.09	1.00	0.17	-	-	-	-	-	8	1	0.13	1.00	0.22	12	1	0.08	1.00	0.15
Composite	1	11	1	0.09	1.00	0.17	1	1	1.00	1.00	1.00	1	1	1.00	1.00	1.00	1	1	1.00	1.00	1.00
Decorator	1	2	1	0.50	1.00	0.67	3	0	0.00	0.00	0.00	3	1	0.33	1.00	0.50	1	1	1.00	1.00	1.00
Factory Method	2	0	0	0.00	0.00	0.00	-	-	-	-	-	2	1	0.50	0.50	0.50	0	0	0.00	0.00	0.00
Observer	2	26	2	0.08	1.00	0.14	-	-	-	-	-	4	0	0.00	0.00	0.00	0	0	0.00	0.00	0.00
Singleton	2	2	2	1.00	1.00	1.00	-	-	-	-	-	2	2	1.00	1.00	1.00	2	2	1.00	1.00	1.00
State/Strategy	6	7	1	0.14	0.17	0.15	-	-	-	-	-	24	2	0.08	0.33	0.13	22	3	0.13	0.50	0.20
Template Method	2	4	2	0.50	1.00	0.67	-	-	-	-	-	5	1	0.20	0.50	0.29	-	-	-	-	-
Total	18	64	11				42	2				61	10				57	9			

11 Gamma's DPs available in this project, since the rest of P-Mart projects cannot provide training instances for the Prototype pattern. Three recent DPD methods are chosen for comparison: DePATOS, which detects structural patterns using a sub-graph isomorphism algorithm (Yu et al., 2018); MLDA, a rule-based approach that analyses method signatures Al-Obeidallah et al. (2018); and SparT, a method based on ontologies that combines structural, behavioural and semantic information Xiong and Li (2019). All these works provide results on JHotDraw that can be contrasted, since their authors both report absolute values of recovered instances, and give access to the DP implementations found. This allows us to compute the performance metrics (precision, recall, and F1) on the basis of a common ground truth, instead of comparing results validated over custom repositories created by the authors. Notice that, usually, P-Mart is extended in DPD studies, but the resulting datasets are not publicly available.

Table 9 lists the DPs under study and the results of the four methods in terms of number of recovered instances per DP, as well as the total sum. Precision, recall and F1 are computed considering positive samples labelled in P-Mart as the ground truth for all methods. None of the methods under comparison distinguish State from Strategy, so we report them together for GEML[7] too. The symbol "-" is used to indicate that the particular DP is not supported by the method. GEML is the method that recovers more true DP implementations (11), followed by SparT (10) and MLDA (9). In terms of F1, GEML is the best method for four DPs (Adapter, Observer, Singleton and Template method), and the second best method for the remaining DPs. The only DP for which no implementation is found is the Factory method, which is not supported by DePATOS and not detected by MLDA. GEML considerably reduces the number of false positives for the Adapter and State/Strategy patterns compared to the rest of methods. In contrast, given that GEML is a ML-based approach, it suffers from the low number of training instances in some cases, such as Composite and Observer. Even so, GEML manages to detect the same number of true positives than the other methods for the Composite pattern, and it is the only method that is able to recover the two Observer implementations.

[7]Notice that GEML can detect both State and Strategy design patterns separately.

8. Experiment 3: Analysis of Applicability

Experiment #3 is explained in this section on the basis of a practical scenario, and evaluated according to more qualitative aspects. The experimentation has been performed with a large number of DPs (15), showing that no adaptation is required to execute GEML when new DPs are introduced. The outcomes serve us to analyse how the change of the training repository might influence the behaviour of the proposed method (RQ4). Then, a comparison against reference non-ML-based DPD tools is provided (RQ5).

8.1. Influence of Training Factors

In response to RQ4, GEML is retrained by applying its general-purpose configuration and using a different input repository than in Experiment #1, as explained in Section 5.3. Then, the detection model is tested on DPExample. Table 10 shows the results for the median and best execution of the 30 runs. *Pr* and *Re* stand for precision and recall, respectively. Design patterns are sorted in decreasing number of positive training samples (see Table 3).

The results bring new insights. Despite the low number of training instances in the repository —a limiting factor inherent to machine learning— GEML is able to retrieve DP instances for 15 design patterns, all Gamma's DP available in P-Mart, with the exception of Builder, Memento, Prototype and Facade. A general observation is that our method suffers when less than 10 training samples are provided. In these cases, it might happen that the lack of data result in executions for which no detection rules can be generated (marked as '-' in Table 10). The degradation in performance is more evident in terms of precision, *i.e.,* the rate of false positives tends to increase. Even so, GEML finds at least half of the true DP implementations during its best execution, with the exception of four design patterns (Observer, Abstract Factory, Command and Factory Method).

Another consequence of having a reduced training set is overfitting, since less variability of examples is presented to the algorithm. The fact that DPExample is the only project taken from the literature, in contrast to other open source projects used for training, may cause this project

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

Table 10: Performance of GEML in the test project (DPExample) with and without using numerical properties

	With numerical properties						**Without numerical properties**					
	Median			Best			Median			Best		
	Pr	Re	F_1	Pr	Re	F_1	Pr	Re	F_1	Pr	Re	F_1
State	0.09	0.67	0.16	0.13	0.67	0.22	0.10	0.67	0.18	0.14	0.67	**0.23**
Adapter	0.04	0.58	0.08	0.12	0.83	0.21	0.04	0.67	0.08	0.17	0.33	**0.22**
Singleton	0.81	0.81	0.81	0.81	0.81	**0.81**	0.81	0.81	0.81	0.81	0.81	**0.81**
Template Method	0.22	0.21	0.20	0.55	0.86	**0.67**	0.40	0.86	0.55	0.55	0.86	**0.67**
Proxy	0.38	0.75	0.50	0.50	0.75	0.60	0.43	0.75	0.55	0.60	0.75	**0.67**
Observer	0.14	0.29	0.19	0.67	0.29	0.40	0.50	0.14	0.22	1.00	0.29	**0.44**
Strategy	0.09	0.50	0.15	0.10	0.50	0.16	0.09	0.50	0.15	0.11	0.63	**0.19**
Composite	0.10	0.50	0.17	0.12	0.50	0.19	0.10	0.50	0.17	0.13	0.67	**0.22**
Abstract Factory	-	-	-	0.04	0.37	0.08	-	-	-	0.16	0.79	**0.25**
Command	-	-	-	0.01	0.40	0.02	0.01	0.40	0.02	0.01	0.20	**0.02**
Factory Method	-	-	-	-	-	-	-	-	-	0.12	0.20	**0.15**
Visitor	0.36	0.93	0.52	0.42	0.93	**0.58**	0.36	0.93	0.52	0.42	0.93	**0.58**
Iterator	0.02	0.40	0.03	0.80	0.80	**0.80**	0.02	0.40	0.03	0.67	0.80	0.73
Decorator	0.75	0.33	0.45	0.80	0.67	**0.73**	1.00	0.33	0.50	0.80	0.67	**0.73**
Bridge	0.04	0.17	0.07	0.07	0.50	**0.13**	0.04	0.17	0.07	0.07	0.50	**0.13**

to present slight differences for some properties relying on role identification and software metrics. This is specially evident for two DPs, Adapter and Template Method, since GEML provides a considerably higher precision and recall in the training phase. For the Adapter, we have observed differences in how DPB and P-Mart label the roles, since DPB assigns one class per role at most (Zanoni et al., 2015). This affects the effectiveness of the *adapterMethod* operator, which works under the same assumption. Similarly, we realised that numerical properties (software metrics) become less informative in this experiment. The difference in size between classes inspected during training and those available for testing implies that the learned thresholds for measures like LOC and NOM are not so representative for the testing project. After excluding such operators, our method has improved its detection capability for six design patterns and has guaranteed more stable results for Abstract Factory, Command and Factory Method (see Table 10). The generated rule set for each DP is available as additional material.

The specificity of some operators has revealed as an important factor to counteract the low number of training samples. This is reflected in the Decorator pattern, for which GEML provides similar results than those obtained in Experiment #1 after learning from two positive samples only in Experiment #2. Looking at the resulting detection models, we observe that *redirectInFamily* — an operator particularly associated to the Decorator pattern — frequently appears in the rules, as shown in Section 6.2. This finding does not imply that new operators oriented towards a particular DP have to be implemented to support its detection. This can be observed in the case of the Visitor pattern, which has obtained the highest recall despite being a pattern with very few positive samples and not being considered in Experiment #1. For this specific pattern, the operator checking whether a role is implemented by an interface becomes highly relevant to identify its structure. Finally, State and Strategy share the same class structure, making it difficult to distinguish them and therefore to detect them correctly. For this reason, some DPD methods consider them together. According to our outcomes, GEML is able to differentiate State from Strategy, and vice versa, for around 10% of instances, and the corresponding false positives are often attributed to an usual misclassification between both DPs.

Finally, Table 11 provides the average execution time of each step of the DPD process applied to DPExample, a medium-size project. G3P4DPD is the step requiring more time, although it does not usually takes longer than one minute. Pruning the generated rules is a very fast procedure that mostly depends on the number of rules returned by G3P4DPD. Note that training the DPD model, *i.e.,* rule generation and pruning, only needs to be carried out when new DP instances are added to the repository. The VF2 algorithm is efficient for candidate generation, and no great disparity is observed for design patterns with different number of roles. Then, the filtering step takes two seconds at most, depending on the different heuristics applied to each DP (see Section 5.3). Finally, the time required to proceed with the detection depends on both the number of candidates and the number of rules.

In light of these results, the influence of the training conditions (RQ4) can be summarised as follows:

- The detection performance of GEML reaches up to 81%, though it varies depending on the design pattern. GEML achieves more than 50% of recall for 11 out of the 15 DPs, and more than 75% for six of them, despite the low number of samples available.

- The detection capability is not only affected by the training set size, but also the particular characteristics of each DP and how roles are labelled in the repository.

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

Table 11: Average execution time (ms) of each phase of the DPD process. Graph building from code takes 909.67 ms on average.

	Rule		Candidates		DP
	gener.	pruning	gener.	filtering	detection
State	68,188.23	29.27	24.00	168.37	1,066.80
Adapter	27,949.63	19.20	22.37	300.87	722.67
Singleton	19,112.73	6.50	15.57	204.23	1,140.40
T. Method	39,412.37	17.83	18.20	178.30	584.30
Proxy	12,099.03	3.33	18.47	160.83	85.83
Observer	5,157.50	67.67	24.50	75.37	60.43
Strategy	6,980.83	1.63	24.03	260.57	197.23
Composite	14,909.90	1.40	22.30	237.47	697.50
A. Factory	8,852.90	1.20	23.20	1,733.53	52.67
Command	11,596.50	0.97	37.30	240.57	342.03
F. Method	5,145.03	0.80	31.20	160.07	2.67
Visitor	9,765.87	0.70	17.13	27.93	3.13
Iterator	3,928.11	0.74	23.95	14.03	110.71
Decorator	4,174.20	0.50	35.43	434.97	3.93
Bridge	3,961.95	0.72	31.55	158.61	4.10

Table 12: Comparison results for Experiment #2

	Ground	GEML		SSA		Ptidej	
	truth	P	TP	P	TP	P	TP
State	12	57	8	41	3	104	9
Adapter	6	12	2	54	3	128	3
Singleton	21	21	17	22	17	82	15
T. Method	7	11	6	20	7	234	7
Proxy	4	5	3	2	2	127	4
Observer	7	2	2	4	2	-	-
Strategy	8	46	5	6	6	0	0
Composite	6	30	2	7	1	29	2
A. Factory	19	99	15	-	-	-	-
Command	5	79	1	4	3	36	2
F. Method	15	26	3	1	1	44	5
Visitor	15	33	14	11	11	2	2
Iterator	5	5	4	-	-	-	-
Decorator	6	5	4	19	5	-	-
Bridge	6	41	3	3	0	-	-
Total	142	472	89	194	61	786	49

- The choice of operators, especially not using software metrics, become more relevant when few samples are available. The improvement can be up to 300% in terms of precision, 114% in terms of recall and 213% with respect to F1.
- GEML is able to learn rules for any new design pattern without requiring the implementation of specific operators, but might have difficulties to produce rules for some DPs when the number of samples is significantly low.

8.2. Comparison with DPD Tools

In the context of RQ3, the previous results should be contrasted with those obtained with other tools available to the software engineer, *i.e.,* SSA and Ptidej. It should be noted that the conditions under which this comparison can be carried out (see Section 5.3) are not favourable for GEML. On the one hand, the number of training samples for some DPs is extremely low. On the other hand, since GEML has partially learned from the outcomes of both tools, GEML might fail to identify the instances not discovered by these tools. Even so, GEML manages to be superior, or at least competitive, to SSA and Ptidej. Table 12 shows the best results for GEML together with those obtained after running both tools. We report absolute numbers of positive DP implementations retrieved (P) and correctly identified (TP) by each method, since they seem easier to interpret and give an idea of the effort required to manually verify tool outcomes. As a reference, the actual number of DP implementations (ground truth) is provided too.

Overall, GEML detects 63% of the DP implementations, followed by SSA (43%) and Ptidej (35%). SSA applies a more conservative detection strategy that allows reducing the presence of false positives (FPs). Proxy, Strategy, Factory Method, Bridge and Visitor clearly illustrate this point. GEML shows more variability in this regard, with more than a half of FPs corresponding to three DPs: Abstract Factory (not supported by the other tools), Command and State. The issues discussed in previous Section 8.1, such as the low number of instances and overfitting, are the reason behind such behaviour. Even so, GEML considerably reduces the need of inspecting all classes within the project, and tends to return less FPs than Ptidej. We observe that this tool correctly identifies the classes implementing the design pattern, but fails to assign the roles and returns several permutations as solution for the same DP instance. SSA presents some limitations with respect to role identification too. For 12 DPs, SSA does not provide the classes playing one or more roles, meaning that the practitioner is required to manually inspect the code to complete the DP definition.

Each tool appears to be superior for a different set of design patterns, not necessarily those belonging to the same category (creational, structural or behavioural). Indeed, the only DPs for which the three tools return the majority of implementations are Singleton and Template Method. Difficulties to detect implementations of Factory Method, Composite and Adapter are observed in the three tools. SSA finds one DP instance more than GEML for Adapter, Template Method, Strategy and Decorator, and two more instances of the Command pattern. Higher differences are observed in favour of GEML for State (5), Bridge (3), Visitor (3) and Factory Method (2). Compared to Ptidej, GEML also returns a similar number of correct DP instances for State, Adapter, Template Method and Proxy. However, the lack of support to four DPs (Observer, Bridge, Abstract Factory and Decorator) and the fact that Strategy and State are considered together impose some limitations to Ptidej.

At this point the level of detection agreement between tools is discussed based on the intersection of their result sets so as not to focus solely on the number of DP instances detected. These coincidences are expressed as percentages in Table 13, what includes outcomes for both positive DP implementations and TPs. In light of the results, it does not seem evident to conclude which pair of tools provide more similar results. Due to the FP rate of Ptidej, GEML is closer to SSA in terms of positive implementations. Nevertheless, GEML achieves higher agreement with Ptidej for State and Proxy compared to the coincidences between SSA and Ptidej.

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

Table 13: Coincidences among DPD tools

	Unique TP	Coincidences in positive implementations				Coincidences in true positive implementations			
		GEML ∩ SSA	GEML ∩ Ptidej	SSA ∩ Ptidej	All	GEML ∩ SSA	GEML ∩ Ptidej	SSA ∩ Ptidej	All
State	1	38.0%	53.3%	33.0%	24.6%	37.5%	70.0%	33.3%	30.0%
Adapter	0	15.8%	6.8%	20.6%	4.2%	75.0%	75.0%	100.0%	75.0%
Singleton	0	95.4%	17.15%	16.9%	16.9%	100.0%	68.4%	68.4%	68.4%
T. Method	0	52.4%	5.1%	8.6%	4.7%	85.7%	85.7%	100.0%	85.7%
Proxy	0	40.0%	1.5%	0.8%	0.8%	66.7%	50.0%	25.0%	25.0%
Observer	2	0.0%	-	-	-	0.0%	-	-	-
Strategy	0	10.6%	-	-	-	83.3%	-	-	-
Composite	2	23.3%	25.5%	12.5%	8.5%	25.0%	50.0%	50.0%	25.0%
A. Factory	15	-	-	-	-	0.0%	-	-	-
Command	0	2.4%	3.6%	8.6%	0.0%	40.0%	0.0%	75.0%	0.0%
F. Method	0	0.0%	40.0%	0.0%	0.0%	0.0%	60.0%	-	0.0%
Visitor	3	78.8%	6.1%	18.2%	6.1%	83.9%	6.5%	18.2%	6.5%
Iterator	4	-	-	-	-	-	-	-	-
Decorator	1	14.3%	-	-	-	50.0%	-	-	-
Bridge	3	7.3%	-	-	-	0.0%	-	-	-

Next, we focus on those design patterns for which tools exhibit less agreement. GEML and SSA both found two out of the seven implementations of the Observer, but they returned different instances. Similarly, the implementation found by SSA for the Factory Method was not detected by GEML. This phenomenon is observed for Proxy implementations too, for which no pair of tools returns more than two equal instances. Furthermore, Table 13 also provides the number of TPs that only GEML was able to find. In total, GEML detects 12 implementations missed by SSA or Ptidej, not counting the Abstract Factory nor the Iterator instances, since neither SSA nor Ptidej consider these DPs. For the Observer, the Bridge and the Composite patterns, these DP implementations are the only TPs returned by GEML, meaning that it is highly effective to find instances that other tools would miss. As for the Visitor, apart from the three unique implementations returned, GEML is able to find all the instances (11) detected by SSA. This is not what happens more often and, even though each tool has its own ability to find out certain DP implementations, in most cases their results are complementary.

To conclude, we next compile the most relevant facts that give response to RQ5:

- GEML finds more true positive instances than the two reference tools, SSA and Ptidej, including samples of the Abstract Factory pattern and Iterator (not available in these tools).

- Considering only those DPs supported by all tools, 11% of the samples recovered by GEML were not found by any other tool used for comparison.

- The level of agreement among all tools, in terms of true positive implementations, can reach more than 80%, but for some DPs is significantly lower. Tools for DPD are mutually complementary in terms of practical use.

9. Demonstration Tool

GEML is publicly available as a Java-based demonstration tool (see Additional Material) that allows engineers to detect DP implementations from their own projects without requiring any expertise in ML or evolutionary techniques. The tool provides basic graphical support for the whole DPD process, divided into the following three phases:

1. *Generation of candidates.* The source code is analysed to extract an initial set of potential DP implementations (candidates).
2. *Learning of the detection model.* The G3P4DPD and pruning algorithms are executed to generate the set of detection rules.
3. *Recovery of design patterns.* The detection rules, together with a classification strategy chosen by the software engineer, check whether the candidates are actually implementing a DP.

Each of these phases is detailed next. Firstly, for the *generation of candidates*, the project from which the set of potential samples for a given DP will be extracted must be selected. During this process, code artefacts and their relationships are scrutinised to find groups of related elements. Then, a role mapping procedure assigns a role to every artefact comprising the candidate according to its relationships. Taking the Adapter pattern as an example, the code artefact playing the *adapter* role has to implement the interface declared by *target*, while adapting the service provided by *adaptee*. Thus, it should be linked to at least two other code artefacts. Similarly, in the case of the Singleton pattern, each code artefact would correspond to a

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

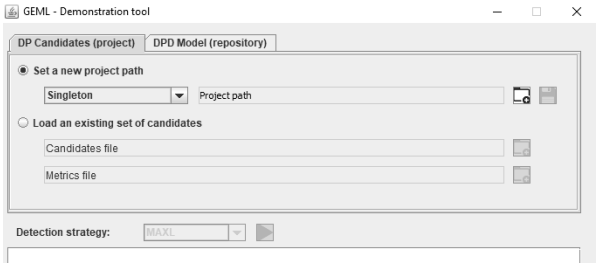


Figure 7: Step 1: Generation of candidates

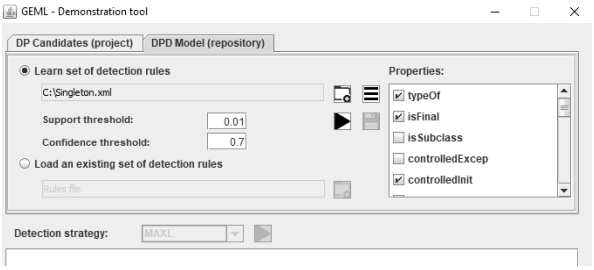


Figure 8: Step 2: Learning of the detection model

```
if
  ctorVisibility(singleton) != public
  and aggregation(singleton,singleton) != notLinked
  and DIT(singleton) < 2
then
  aPattern

if
  ctorVisibility(singleton) = public
  and controlledExcept(singleton) = false
  and controlledInit(singleton) = false
then
  notAPattern
```

Figure 9: Two sample rules generated by the G3P4DPD algorithm

Figure 10: Step 3: Recovery of Singleton pattern instances

DP candidate, as it only has one role. Fig. 7 shows the screenshot with this configuration panel. Once candidates are found, they can be exported for future executions. A file containing software metrics is also generated.

In a different view, the *learning of the detection model* is conducted, the support and confidence thresholds being set as shown in Fig. 8. The user could also select the set of operators to be applied for mining the rules (see Sections 4.1.2 and 6.2). The rest of parameters are set to their default values for simplicity, though advanced users could still modify them by simply editing an XML configuration file. This file also includes the path of the repository from which rules will be mined, so the user could modify it according to organisational or team requirements. Additionally, the resulting detection rules can be saved for future use.

To illustrate this operation, we have selected the project datapro4j[8], a Java library for processing and handling data from heterogeneous data sources, which contains implementations of the Singleton pattern. Fig. 9 shows two illustrative rules returned by G3P4DPD: one describing positive samples —consequent *aPattern*— and one for negative samples —consequent *notAPattern*. On the one hand, the first rule implies that implementations of the Singleton pattern scrutinised from this repository contain a non-public constructor, and a property of their own type to ensure that only one instance is created. In addition, DIT values are not greater than one, meaning that Singleton instances have no superclasses in this repository, except for *Object*, as any Java class. On the other hand, the second rule determines that those classes with a public constructor whose double invocation is not controlled by exceptions are not a valid implementation of Singleton. Notice that the use of exceptions is an alternative implementation of Singleton, which is usually coded in terms of a private constructor whose invocation is controlled by a static variable. However, the use of design microstructures as grammar operators allows detecting both cases. For this execution, the support and confidence thresholds were set to 0.01 and 0.7. In addition, the list of operators were limited to only those that best represent the Singleton pattern (see Section 6.2).

Finally, as for the *recovery of design patterns*, the classification strategy has to be selected. For the datapro4j example, we applied the MAXL strategy. Fig. 10 shows the implementations found for this pattern and repository, which are highly coincident with the actual specification of the library. It is also worth noting that these implementations could be added to the repository in order to gradually adjust this database to the corporate culture in future detections.

10. Threats to Validity

Internal threats are those related to aspects of the experimentation that cannot ensure the causality of the obtained results. Here, the stochastic nature of the algorithm, as well as its setup and parametrisation are internal threats to be considered. Therefore, all experiments are based on 30 independent executions. Furthermore, a parameter study was conducted to determine the best values. As for the construction of the detection model, a stratified 10-fold cross validation is performed to avoid any bias due to the training data. Another threat for the internal validity refers to the setup of the detection method and, more

[8] *Datapro4j*, available from `http://www.jrromero.net/tool_datapro4j.html` (accessed June 22, 2020)

Published version available at: `https://doi.org/10.1016/j.jss.2021.110919`

specifically, to the selection of the pruning and classification strategies. On the one hand, the database coverage method has been considered for pruning, as it is a well-known, proved method in the AC literature, and has been already applied jointly to four of the classification strategies considered in this study. On the other hand, the detection performance of several classification strategies has been analysed as part of the experimentation. The statistical analysis has shown that the detection model is not greatly affected by these algorithms.

External validity is related to the generalisation of the experimental results. In this approach, we have selected 15 design patterns for validation and comparison purposes. In addition, these patterns present different number of roles and multiple roles played by a single artefact. Even so, the low number of training samples available for some DPs could limit the generalisation of conclusions. Similarly, the use of DPB and P-Mart repositories, which contain Java projects, implies that our conclusions could not be directly extrapolated to organisational environments of different nature. Nevertheless, these projects are of non-trivial size and they have become frequent benchmarks within the field. The application of GEML to other industrial projects might require its adaptation to their specific requirements, *e.g.,* other programming languages or design microstructures (grammar operators). In this sense, the possibility to extend the collection of operators and the flexibility provided by the CFG makes GEML adaptable to organisational changes.

11. Concluding Remarks

GEML was introduced as a novel automatic approach for design pattern detection based on evolutionary machine learning. Knowledge from code repositories is extracted by means of G3P4DPD in form of association rules, a highly readable format to represent knowledge (Grosan and Abraham, 2011). The use of an extendable context-free grammar to declare the syntax of rules makes the learning process highly flexible and adaptable to new organisational environments and design patterns. The application of a pruning method and several classification strategies — already proved in well-known associative classification approaches — to select those rules with best detection capabilities leads GEML to accurate and robust predictions.

An extensive experimentation shows that GEML is able to generate high-quality rules describing implementations of structural, creational and behavioural design patterns. A first study reveals that the prediction performance remains robust — also improving in general terms to other related proposal — even when a single common parametrisation is used for all the design patterns, without the need to adjust it individually. We think that this fact could significantly increase its application in practice, since the software engineer is not required to adjust the parameters. We have also analysed whether the detection capabilities of GEML are affected by changes in the training conditions, using a total of 15 DPs, including behavioural, creational and structural patterns. This is the largest set of patterns analysed so far for a ML-based DPD proposal. Results reveal that GEML remains highly competitive even when very few DP examples are available for learning. Compared to reference DPD methods and tools, GEML supports additional DPs and detects more implementations, although the low number of training samples could cause some variability in the rate of false positives. A demonstration tool is provided to show its use and allow programmers to analyse their own Java projects.

In the future we plan to incorporate new design microstructures — in form of grammar operators — that facilitate the detection of the remaining DPs. In addition, the current tool can be evolved and integrated within existing IDE platforms like Eclipse, thus integrating detection capabilities as part of the programming and maintenance tasks.

Additional Material

For replicability purposes, experimentation data, such as the generated DP instances, are available for download, as well as the results from the experimentation and statistical analysis, and the demonstration tool, from `https://www.uco.es/kdis/sbse/geml/`

Acknowledgements

This work was supported by the Spanish Ministry of Economy and Competitiveness [project TIN2017-83445-P], the Spanish Ministry of Education under the FPU program [grant FPU17/00799] and the University of Córdoba [postdoctoral grant "Plan propio - mod. 2.4"].

Appendix A. Extended results for the parameter study

Tables A.14- A.18 compile the results for all combinations of classification strategy, support and confidence thresholds, for the Singleton, Adapter, Factory Method and Composite patterns, respectively. Values represent the average and the standard deviation of each performance measure, i.e. accuracy, precision, recall, specificity and F_1. Bold typeface is used to highlight the best result for the respective classification strategy, and shaded cells represent the global best value for each measure.

References

References

Agrawal, R., Imieliński, T., Swami, A., 1993. Mining Association Rules Between Sets of Items in Large Databases. In: Proc. 1993 ACM SIGMOD Int. Conf. Management of Data. pp. 207–216.

Published version available at: `https://doi.org/10.1016/j.jss.2021.110919`

Table A.14: Classification performance for the Singleton

	Supp - Conf	Accuracy	Precision	Recall	Specificity	F_1
MAXL	(0.01) - (0.5)	0.9395 ± 0.0140	0.9436 ± 0.0164	0.9008 ± 0.0311	0.9631 ± 0.0115	0.9152 ± 0.0222
	(0.01) - (0.6)	0.9434 ± 0.0125	0.9524 ± 0.0163	0.9029 ± 0.0262	0.9681 ± 0.0111	0.9200 ± 0.0187
	(0.01) - (0.7)	**0.9458 ± 0.0123**	**0.9535 ± 0.0140**	**0.9084 ± 0.0288**	**0.9685 ± 0.0106**	**0.9246 ± 0.0184**
	(0.05) - (0.5)	0.9317 ± 0.0171	0.9427 ± 0.0195	0.8821 ± 0.0327	0.9617 ± 0.0137	0.9038 ± 0.0265
	(0.05) - (0.6)	0.9190 ± 0.0142	0.9434 ± 0.0200	0.8441 ± 0.0279	0.9641 ± 0.0135	0.8820 ± 0.0208
	(0.05) - (0.7)	0.9197 ± 0.0122	0.9422 ± 0.0195	0.8467 ± 0.0228	0.9640 ± 0.0130	0.8832 ± 0.0187
	(0.1) - (0.5)	0.9373 ± 0.0140	0.9461 ± 0.0175	0.8916 ± 0.0293	0.9652 ± 0.0104	0.9119 ± 0.0206
	(0.1) - (0.6)	0.9254 ± 0.0107	0.9493 ± 0.0175	0.8550 ± 0.0230	0.9678 ± 0.0105	0.8912 ± 0.0165
	(0.1) - (0.7)	0.9225 ± 0.0086	0.9458 ± 0.0148	0.8508 ± 0.0183	0.9660 ± 0.0106	0.8879 ± 0.0117
DFML	(0.01) - (0.5)	0.9431 ± 0.0132	0.9495 ± 0.0134	0.9038 ± 0.0353	0.9670 ± 0.0097	0.9197 ± 0.0213
	(0.01) - (0.6)	0.9454 ± 0.0118	0.9493 ± 0.0143	0.9126 ± 0.0234	0.9654 ± 0.0105	0.9239 ± 0.0173
	(0.01) - (0.7)	**0.9490 ± 0.0143**	**0.9539 ± 0.0143**	0.9162 ± 0.0309	**0.9689 ± 0.0110**	**0.9290 ± 0.0220**
	(0.05) - (0.5)	0.9291 ± 0.0186	0.9166 ± 0.0299	0.9083 ± 0.0288	0.9418 ± 0.0237	0.9054 ± 0.0242
	(0.05) - (0.6)	0.9229 ± 0.0129	0.9370 ± 0.0219	0.8614 ± 0.0195	0.9599 ± 0.0147	0.8897 ± 0.0179
	(0.05) - (0.7)	0.9223 ± 0.0123	0.9382 ± 0.0226	0.8586 ± 0.0171	0.9612 ± 0.0152	0.8894 ± 0.0165
	(0.1) - (0.5)	0.9204 ± 0.0167	0.8906 ± 0.0268	**0.9189 ± 0.0214**	0.9214 ± 0.0219	0.8980 ± 0.0202
	(0.1) - (0.6)	0.9263 ± 0.0116	0.9350 ± 0.0213	0.8754 ± 0.0211	0.9569 ± 0.0154	0.8961 ± 0.0166
	(0.1) - (0.7)	0.9266 ± 0.0088	0.9414 ± 0.0167	0.8669 ± 0.0132	0.9628 ± 0.0118	0.8952 ± 0.0116
DFML_{χ^2}	(0.01) - (0.5)	0.9429 ± 0.0116	0.9320 ± 0.0185	0.9266 ± 0.0296	0.9531 ± 0.0144	0.9227 ± 0.0175
	(0.01) - (0.6)	0.9520 ± 0.0112	0.9409 ± 0.0145	0.9407 ± 0.0238	0.9589 ± 0.0113	0.9351 ± 0.0173
	(0.01) - (0.7)	**0.9561 ± 0.0136**	**0.9460 ± 0.0147**	**0.9461 ± 0.0298**	0.9621 ± 0.0118	**0.9411 ± 0.0202**
	(0.05) - (0.5)	0.9404 ± 0.0117	0.9346 ± 0.0163	0.9181 ± 0.0249	0.9539 ± 0.0132	0.9192 ± 0.0175
	(0.05) - (0.6)	0.9253 ± 0.0106	0.9347 ± 0.0183	0.8712 ± 0.0199	0.9577 ± 0.0126	0.8943 ± 0.0157
	(0.05) - (0.7)	0.9220 ± 0.0106	0.9295 ± 0.0202	0.8680 ± 0.0184	0.9550 ± 0.0134	0.8909 ± 0.0151
	(0.1) - (0.5)	0.9433 ± 0.0116	0.9372 ± 0.0192	0.9207 ± 0.0186	0.9571 ± 0.0130	0.9234 ± 0.0162
	(0.1) - (0.6)	0.9282 ± 0.0103	0.9359 ± 0.0187	0.8779 ± 0.0169	0.9586 ± 0.0121	0.8988 ± 0.0145
	(0.1) - (0.7)	0.9284 ± 0.0094	0.9420 ± 0.0151	0.8726 ± 0.0156	**0.9623 ± 0.0103**	0.8979 ± 0.0131
DFML_{Lap}	(0.01) - (0.5)	0.9278 ± 0.0129	0.9430 ± 0.0136	0.8690 ± 0.0317	0.9635 ± 0.0084	0.8960 ± 0.0216
	(0.01) - (0.6)	0.9370 ± 0.0130	0.9489 ± 0.0145	0.8894 ± 0.0256	0.9660 ± 0.0107	0.9103 ± 0.0193
	(0.01) - (0.7)	**0.9422 ± 0.0130**	0.9498 ± 0.0174	**0.9028 ± 0.0275**	0.9661 ± 0.0127	**0.9191 ± 0.0192**
	(0.05) - (0.5)	0.9265 ± 0.0134	0.9487 ± 0.0195	0.8613 ± 0.0282	0.9658 ± 0.0136	0.8933 ± 0.0212
	(0.05) - (0.6)	0.9125 ± 0.0133	0.9448 ± 0.0201	0.8240 ± 0.0270	0.9657 ± 0.0130	0.8707 ± 0.0210
	(0.05) - (0.7)	0.9189 ± 0.0126	0.9452 ± 0.0196	0.8400 ± 0.0244	0.9668 ± 0.0114	0.8815 ± 0.0195
	(0.1) - (0.5)	0.9329 ± 0.0129	0.9529 ± 0.0165	0.8719 ± 0.0308	0.9700 ± 0.0094	0.9032 ± 0.0199
	(0.1) - (0.6)	0.9198 ± 0.0111	**0.9542 ± 0.0136**	0.8343 ± 0.0263	0.9711 ± 0.0084	0.8803 ± 0.0180
	(0.1) - (0.7)	0.9182 ± 0.0111	0.9532 ± 0.0127	0.8307 ± 0.0275	**0.9713 ± 0.0084**	0.8785 ± 0.0175

Table A.15: Classification performance for the Adapter

	Supp - Conf	Accuracy	Precision	Recall	Specificity	F_1
MAXL	(0.01) - (0.5)	0.8542 ± 0.0047	0.8356 ± 0.0051	0.8885 ± 0.0066	0.8190 ± 0.0059	0.8604 ± 0.0047
	(0.01) - (0.6)	0.8624 ± 0.0034	0.8402 ± 0.0035	0.9021 ± 0.0058	0.8218 ± 0.0044	0.8692 ± 0.0034
	(0.01) - (0.7)	0.8657 ± 0.0028	**0.8431 ± 0.0029**	0.9048 ± 0.0047	**0.8258 ± 0.0037**	0.8722 ± 0.0028
	(0.05) - (0.5)	0.8604 ± 0.0024	0.8411 ± 0.0023	0.8953 ± 0.0039	0.8247 ± 0.0027	0.8666 ± 0.0024
	(0.05) - (0.6)	0.8630 ± 0.0025	0.8411 ± 0.0029	0.9012 ± 0.0046	0.8237 ± 0.0039	0.8695 ± 0.0025
	(0.05) - (0.7)	0.8642 ± 0.0026	0.8419 ± 0.0029	0.9028 ± 0.0047	0.8246 ± 0.0040	0.8706 ± 0.0026
	(0.1) - (0.5)	0.8667 ± 0.0015	0.8402 ± 0.0013	0.9121 ± 0.0029	0.8203 ± 0.0013	0.8740 ± 0.0017
	(0.1) - (0.6)	0.8672 ± 0.0018	0.8407 ± 0.0021	**0.9124 ± 0.0024**	0.8208 ± 0.0026	0.8744 ± 0.0017
	(0.1) - (0.7)	**0.8675 ± 0.0019**	0.8413 ± 0.0030	0.9120 ± 0.0024	0.8220 ± 0.0032	**0.8746 ± 0.0018**
DFML	(0.01) - (0.5)	0.8621 ± 0.0045	0.8392 ± 0.0046	0.9020 ± 0.0073	0.8211 ± 0.0061	0.8686 ± 0.0046
	(0.01) - (0.6)	0.8641 ± 0.0041	**0.8464 ± 0.0044**	0.8964 ± 0.0083	**0.8309 ± 0.0059**	0.8697 ± 0.0044
	(0.01) - (0.7)	**0.8679 ± 0.0028**	0.8450 ± 0.0040	0.9069 ± 0.0057	0.8279 ± 0.0059	**0.8743 ± 0.0028**
	(0.05) - (0.5)	0.8636 ± 0.0028	0.8418 ± 0.0022	0.9021 ± 0.0071	0.8242 ± 0.0036	0.8701 ± 0.0033
	(0.05) - (0.6)	0.8634 ± 0.0026	0.8447 ± 0.0044	0.8969 ± 0.0069	0.8292 ± 0.0061	0.8692 ± 0.0029
	(0.05) - (0.7)	0.8665 ± 0.0033	0.8428 ± 0.0037	0.9069 ± 0.0060	0.8250 ± 0.0051	0.8730 ± 0.0033
	(0.1) - (0.5)	0.8655 ± 0.0024	0.8406 ± 0.0024	0.9084 ± 0.0054	0.8214 ± 0.0031	0.8724 ± 0.0026
	(0.1) - (0.6)	0.8632 ± 0.0033	0.8440 ± 0.0038	0.8974 ± 0.0073	0.8283 ± 0.0057	0.8691 ± 0.0035
	(0.1) - (0.7)	0.8662 ± 0.0026	0.8414 ± 0.0032	**0.9084 ± 0.0047**	0.8230 ± 0.0036	0.8730 ± 0.0028
DFML_{χ^2}	(0.01) - (0.5)	0.8673 ± 0.0013	0.8401 ± 0.0018	0.9134 ± 0.0017	0.8201 ± 0.0019	0.8746 ± 0.0013
	(0.01) - (0.6)	0.8680 ± 0.0022	0.8423 ± 0.0027	0.9119 ± 0.0033	0.8229 ± 0.0030	0.8750 ± 0.0022
	(0.01) - (0.7)	**0.8688 ± 0.0022**	**0.8430 ± 0.0028**	0.9121 ± 0.0032	**0.8244 ± 0.0038**	**0.8757 ± 0.0020**
	(0.05) - (0.5)	0.8670 ± 0.0013	0.8400 ± 0.0015	0.9130 ± 0.0018	0.8198 ± 0.0016	0.8744 ± 0.0013
	(0.05) - (0.6)	0.8671 ± 0.0019	0.8406 ± 0.0025	0.9119 ± 0.0024	0.8212 ± 0.0032	0.8742 ± 0.0018
	(0.05) - (0.7)	0.8684 ± 0.0016	0.8421 ± 0.0027	0.9129 ± 0.0026	0.8229 ± 0.0036	0.8754 ± 0.0014
	(0.1) - (0.5)	0.8674 ± 0.0004	0.8401 ± 0.0013	**0.9137 ± 0.0010**	0.8198 ± 0.0010	0.8748 ± 0.0009
	(0.1) - (0.6)	0.8676 ± 0.0016	0.8407 ± 0.0021	0.9133 ± 0.0017	0.8208 ± 0.0025	0.8748 ± 0.0014
	(0.1) - (0.7)	0.8679 ± 0.0019	0.8414 ± 0.0030	0.9127 ± 0.0022	0.8220 ± 0.0032	0.8750 ± 0.0018
DFML_{Lap}	(0.01) - (0.5)	0.8544 ± 0.0038	0.8388 ± 0.0035	0.8840 ± 0.0070	0.8241 ± 0.0044	0.8600 ± 0.0040
	(0.01) - (0.6)	0.8629 ± 0.0031	0.8416 ± 0.0032	0.9008 ± 0.0063	0.8240 ± 0.0043	0.8694 ± 0.0034
	(0.01) - (0.7)	0.8656 ± 0.0028	**0.8442 ± 0.0027**	0.9025 ± 0.0046	**0.8277 ± 0.0034**	0.8718 ± 0.0028
	(0.05) - (0.5)	0.8603 ± 0.0025	0.8410 ± 0.0023	0.8950 ± 0.0040	0.8246 ± 0.0027	0.8664 ± 0.0026
	(0.05) - (0.6)	0.8638 ± 0.0024	0.8416 ± 0.0029	0.9023 ± 0.0050	0.8243 ± 0.0039	0.8703 ± 0.0025
	(0.05) - (0.7)	0.8641 ± 0.0024	0.8423 ± 0.0027	0.9020 ± 0.0050	0.8252 ± 0.0038	0.8704 ± 0.0025
	(0.1) - (0.5)	0.8666 ± 0.0015	0.8402 ± 0.0013	0.9119 ± 0.0028	0.8202 ± 0.0012	0.8739 ± 0.0017
	(0.1) - (0.6)	0.8672 ± 0.0019	0.8407 ± 0.0022	**0.9124 ± 0.0024**	0.8209 ± 0.0027	**0.8744 ± 0.0018**
	(0.1) - (0.7)	**0.8673 ± 0.0019**	0.8413 ± 0.0029	0.9113 ± 0.0027	0.8221 ± 0.0031	0.8743 ± 0.0019

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

Table A.16: Classification performance for the Factory Method

	Supp - Conf	Accuracy	Precision	Recall	Specificity	F_1
MAXL	(0.01) - (0.5)	0.8200 ± 0.0086	**0.8331 ± 0.0089**	0.8348 ± 0.0099	0.8028 ± 0.0123	0.8329 ± 0.0079
	(0.01) - (0.6)	0.8198 ± 0.0081	0.8294 ± 0.0088	0.8404 ± 0.0110	0.7957 ± 0.0127	0.8335 ± 0.0077
	(0.01) - (0.7)	**0.8266 ± 0.0081**	0.8311 ± 0.0082	0.8540 ± 0.0135	0.7946 ± 0.0119	0.8411 ± 0.0081
	(0.05) - (0.5)	0.7946 ± 0.0074	0.8313 ± 0.0088	0.7790 ± 0.0132	**0.8128 ± 0.0136**	0.8026 ± 0.0076
	(0.05) - (0.6)	0.8028 ± 0.0098	0.8257 ± 0.0102	0.8062 ± 0.0156	0.7988 ± 0.0142	0.8141 ± 0.0099
	(0.05) - (0.7)	0.8156 ± 0.0079	0.8260 ± 0.0076	0.8356 ± 0.0173	0.7922 ± 0.0131	0.8292 ± 0.0085
	(0.1) - (0.5)	0.8102 ± 0.0110	0.8070 ± 0.0128	0.8556 ± 0.0117	0.7573 ± 0.0216	0.8291 ± 0.0091
	(0.1) - (0.6)	0.8174 ± 0.0087	0.8095 ± 0.0087	0.8691 ± 0.0126	0.7572 ± 0.0148	0.8366 ± 0.0081
	(0.1) - (0.7)	0.8221 ± 0.0116	0.7998 ± 0.0122	**0.8992 ± 0.0130**	0.7322 ± 0.0213	**0.8451 ± 0.0097**
DFML	(0.01) - (0.5)	0.8124 ± 0.0089	0.8013 ± 0.0096	0.8697 ± 0.0121	0.7456 ± 0.0159	0.8328 ± 0.0080
	(0.01) - (0.6)	0.8198 ± 0.0099	0.8141 ± 0.0118	0.8663 ± 0.0157	0.7656 ± 0.0186	0.8378 ± 0.0092
	(0.01) - (0.7)	**0.8275 ± 0.0077**	**0.8177 ± 0.0075**	0.8783 ± 0.0175	**0.7683 ± 0.0132**	**0.8455 ± 0.0082**
	(0.05) - (0.5)	0.7958 ± 0.0091	0.7926 ± 0.0115	0.8457 ± 0.0126	0.7376 ± 0.0189	0.8166 ± 0.0079
	(0.05) - (0.6)	0.8069 ± 0.0109	0.8047 ± 0.0099	0.8505 ± 0.0163	0.7560 ± 0.0150	0.8253 ± 0.0105
	(0.05) - (0.7)	0.8219 ± 0.0080	0.8115 ± 0.0084	0.8753 ± 0.0149	0.7597 ± 0.0142	0.8408 ± 0.0080
	(0.1) - (0.5)	0.7908 ± 0.0089	0.7666 ± 0.0098	0.8853 ± 0.0144	0.6805 ± 0.0185	0.8197 ± 0.0081
	(0.1) - (0.6)	0.8062 ± 0.0099	0.7841 ± 0.0102	0.8886 ± 0.0137	0.7100 ± 0.0171	0.8315 ± 0.0088
	(0.1) - (0.7)	0.8143 ± 0.0145	0.7887 ± 0.0125	**0.9001 ± 0.0175**	0.7143 ± 0.0194	0.8393 ± 0.0130
$DFML_{\chi^2}$	(0.01) - (0.5)	0.8201 ± 0.0082	**0.8202 ± 0.0092**	0.8561 ± 0.0113	**0.7780 ± 0.0151**	0.8363 ± 0.0074
	(0.01) - (0.6)	0.8221 ± 0.0090	0.8180 ± 0.0094	0.8655 ± 0.0176	0.7716 ± 0.0153	0.8392 ± 0.0090
	(0.01) - (0.7)	**0.8308 ± 0.0089**	0.8180 ± 0.0071	0.8863 ± 0.0218	0.7661 ± 0.0135	0.8489 ± 0.0099
	(0.05) - (0.5)	0.8102 ± 0.0096	0.8120 ± 0.0084	0.8466 ± 0.0160	0.7677 ± 0.0133	0.8271 ± 0.0095
	(0.05) - (0.6)	0.8197 ± 0.0117	0.8105 ± 0.0098	0.8725 ± 0.0195	0.7582 ± 0.0153	0.8382 ± 0.0114
	(0.05) - (0.7)	0.8304 ± 0.0082	0.8113 ± 0.0081	0.8965 ± 0.0158	0.7533 ± 0.0145	**0.8503 ± 0.0081**
	(0.1) - (0.5)	0.8134 ± 0.0102	0.7910 ± 0.0111	0.8943 ± 0.0119	0.7190 ± 0.0205	0.8377 ± 0.0083
	(0.1) - (0.6)	0.8227 ± 0.0088	0.7971 ± 0.0093	0.9046 ± 0.0124	0.7271 ± 0.0163	0.8461 ± 0.0076
	(0.1) - (0.7)	0.8243 ± 0.0111	0.7940 ± 0.0123	**0.9154 ± 0.0134**	0.7180 ± 0.0228	0.8490 ± 0.0091
$DFML_{Lap}$	(0.01) - (0.5)	0.8097 ± 0.0076	**0.8442 ± 0.0066**	0.7957 ± 0.0125	0.8262 ± 0.0090	0.8177 ± 0.0079
	(0.01) - (0.6)	0.8097 ± 0.0101	0.8325 ± 0.0102	0.8120 ± 0.0137	0.8070 ± 0.0141	0.8206 ± 0.0098
	(0.01) - (0.7)	0.8189 ± 0.0097	0.8355 ± 0.0100	0.8298 ± 0.0149	0.8062 ± 0.0137	0.8310 ± 0.0096
	(0.05) - (0.5)	0.7857 ± 0.0094	0.8315 ± 0.0088	0.7582 ± 0.0195	**0.8177 ± 0.0140**	0.7911 ± 0.0110
	(0.05) - (0.6)	0.7984 ± 0.0093	0.8280 ± 0.0114	0.7934 ± 0.0156	0.8042 ± 0.0160	0.8083 ± 0.0096
	(0.05) - (0.7)	0.8125 ± 0.0084	0.8281 ± 0.0099	0.8256 ± 0.0178	0.7973 ± 0.0159	0.8251 ± 0.0089
	(0.1) - (0.5)	0.8048 ± 0.0118	0.8018 ± 0.0149	0.8532 ± 0.0162	0.7483 ± 0.0271	0.8245 ± 0.0097
	(0.1) - (0.6)	0.8145 ± 0.0093	0.8080 ± 0.0105	0.8657 ± 0.0156	0.7548 ± 0.0196	0.8338 ± 0.0086
	(0.1) - (0.7)	**0.8208 ± 0.0113**	0.7984 ± 0.0128	**0.8993 ± 0.0141**	0.7292 ± 0.0238	**0.8442 ± 0.0092**

Table A.17: Classification performance for the Decorator

	Supp - Conf	Accuracy	Precision	Recall	Specificity	F_1
MAXL	(0.01) - (0.5)	0.7848 ± 0.0135	0.8466 ± 0.0288	0.5449 ± 0.0265	0.9303 ± 0.0132	0.6448 ± 0.0236
	(0.01) - (0.6)	0.7904 ± 0.0161	0.8369 ± 0.0393	0.5605 ± 0.0369	0.9301 ± 0.0141	0.6560 ± 0.0331
	(0.01) - (0.7)	0.7973 ± 0.0168	0.8311 ± 0.0278	**0.5898 ± 0.0393**	0.9232 ± 0.0129	0.6755 ± 0.0341
	(0.05) - (0.5)	0.7882 ± 0.0125	0.8753 ± 0.0335	0.5235 ± 0.0301	0.9489 ± 0.0134	0.6357 ± 0.0279
	(0.05) - (0.6)	0.7936 ± 0.0130	**0.8778 ± 0.0271**	0.5388 ± 0.0312	0.9484 ± 0.0125	0.6530 ± 0.0252
	(0.05) - (0.7)	**0.8051 ± 0.0141**	0.8717 ± 0.0236	0.5779 ± 0.0364	0.9431 ± 0.0105	**0.6794 ± 0.0287**
	(0.1) - (0.5)	0.7690 ± 0.0155	0.8679 ± 0.0341	0.4693 ± 0.0379	**0.9509 ± 0.0140**	0.5913 ± 0.0370
	(0.1) - (0.6)	0.7725 ± 0.0147	0.8592 ± 0.0322	0.4904 ± 0.0386	0.9438 ± 0.0148	0.6064 ± 0.0327
	(0.1) - (0.7)	0.7927 ± 0.0129	0.8526 ± 0.0307	0.5547 ± 0.0319	0.9372 ± 0.0117	0.6584 ± 0.0275
DFML	(0.01) - (0.5)	0.8084 ± 0.0149	0.8552 ± 0.0249	0.6135 ± 0.0327	0.9267 ± 0.0122	0.6968 ± 0.0299
	(0.01) - (0.6)	0.8128 ± 0.0149	0.8508 ± 0.0244	0.6279 ± 0.0322	0.9253 ± 0.0141	0.7068 ± 0.0279
	(0.01) - (0.7)	0.8168 ± 0.0162	0.8393 ± 0.0239	0.6523 ± 0.0399	0.9166 ± 0.0145	0.7203 ± 0.0328
	(0.05) - (0.5)	0.8097 ± 0.0146	0.8424 ± 0.0253	0.6242 ± 0.0375	0.9223 ± 0.0138	0.7018 ± 0.0291
	(0.05) - (0.6)	0.8104 ± 0.0147	0.8514 ± 0.0286	0.6204 ± 0.0318	0.9260 ± 0.0136	0.7039 ± 0.0250
	(0.05) - (0.7)	**0.8212 ± 0.0210**	0.8438 ± 0.0310	**0.6596 ± 0.0489**	0.9193 ± 0.0163	**0.7262 ± 0.0386**
	(0.1) - (0.5)	0.7839 ± 0.0111	**0.8620 ± 0.0290**	0.5219 ± 0.0273	**0.9429 ± 0.0138**	0.6310 ± 0.0248
	(0.1) - (0.6)	0.7851 ± 0.0166	0.8574 ± 0.0365	0.5360 ± 0.0425	0.9363 ± 0.0162	0.6381 ± 0.0372
	(0.1) - (0.7)	0.8117 ± 0.0094	0.8479 ± 0.0239	0.6246 ± 0.0306	0.9254 ± 0.0128	0.7061 ± 0.0210
$DFML_{\chi^2}$	(0.01) - (0.5)	0.8143 ± 0.0174	0.7993 ± 0.0285	0.7037 ± 0.0350	0.8815 ± 0.0189	0.7352 ± 0.0280
	(0.01) - (0.6)	0.8191 ± 0.0141	0.8042 ± 0.0258	0.7099 ± 0.0264	0.8855 ± 0.0170	0.7416 ± 0.0229
	(0.01) - (0.7)	0.8188 ± 0.0150	0.7979 ± 0.0229	0.7184 ± 0.0340	0.8797 ± 0.0155	0.7448 ± 0.0265
	(0.05) - (0.5)	0.8118 ± 0.0141	0.7835 ± 0.0203	0.7122 ± 0.0318	0.8722 ± 0.0167	0.7360 ± 0.0221
	(0.05) - (0.6)	0.8151 ± 0.0150	0.7993 ± 0.0248	0.7059 ± 0.0311	0.8815 ± 0.0173	0.7384 ± 0.0238
	(0.05) - (0.7)	**0.8229 ± 0.0179**	0.8043 ± 0.0269	**0.7251 ± 0.0393**	0.8824 ± 0.0171	**0.7501 ± 0.0302**
	(0.1) - (0.5)	0.8009 ± 0.0123	0.8088 ± 0.0250	0.6379 ± 0.0264	0.8997 ± 0.0157	0.6991 ± 0.0217
	(0.1) - (0.6)	0.8056 ± 0.0162	**0.8208 ± 0.0266**	0.6391 ± 0.0393	**0.9067 ± 0.0157**	0.7031 ± 0.0304
	(0.1) - (0.7)	0.8220 ± 0.0164	0.8166 ± 0.0175	0.7014 ± 0.0377	0.8952 ± 0.0116	0.7428 ± 0.0278
$DFML_{Lap}$	(0.01) - (0.5)	0.7816 ± 0.0127	0.8782 ± 0.0270	0.5031 ± 0.0246	0.9508 ± 0.0119	0.6217 ± 0.0213
	(0.01) - (0.6)	0.7833 ± 0.0150	0.8680 ± 0.0448	0.5049 ± 0.0323	0.9524 ± 0.0128	0.6212 ± 0.0329
	(0.01) - (0.7)	0.7950 ± 0.0172	0.8699 ± 0.0306	0.5489 ± 0.0414	0.9443 ± 0.0138	0.6549 ± 0.0382
	(0.05) - (0.5)	0.7817 ± 0.0147	**0.8850 ± 0.0410**	0.4929 ± 0.0320	**0.9570 ± 0.0150**	0.6139 ± 0.0325
	(0.05) - (0.6)	0.7843 ± 0.0167	0.8838 ± 0.0393	0.5037 ± 0.0370	0.9548 ± 0.0157	0.6251 ± 0.0322
	(0.05) - (0.7)	**0.7999 ± 0.0142**	0.8800 ± 0.0277	**0.5547 ± 0.0336**	0.9488 ± 0.0120	**0.6637 ± 0.0286**
	(0.1) - (0.5)	0.7614 ± 0.0137	0.8739 ± 0.0343	0.4412 ± 0.0357	0.9559 ± 0.0131	0.5675 ± 0.0366
	(0.1) - (0.6)	0.7679 ± 0.0162	0.8722 ± 0.0372	0.4639 ± 0.0377	0.9527 ± 0.0138	0.5870 ± 0.0361
	(0.1) - (0.7)	0.7824 ± 0.0155	0.8552 ± 0.0362	0.5174 ± 0.0356	0.9433 ± 0.0141	0.6307 ± 0.0337

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

Table A.18: Classification performance for the Composite

	Supp - Conf	Accuracy	Precision	Recall	Specificity	F_1
MAXL	(0.01) - (0.5)	0.8718 ± 0.0193	0.8201 ± 0.0791	**0.5544 ± 0.0712**	0.9690 ± 0.0139	0.6309 ± 0.0668
	(0.01) - (0.6)	0.8694 ± 0.0155	0.8109 ± 0.1024	0.5533 ± 0.0491	0.9662 ± 0.0116	0.6244 ± 0.0590
	(0.01) - (0.7)	0.8626 ± 0.0152	0.7976 ± 0.0931	0.5122 ± 0.0513	0.9700 ± 0.0151	0.5935 ± 0.0510
	(0.05) - (0.5)	0.8677 ± 0.0171	0.8211 ± 0.0795	0.5311 ± 0.0632	0.9706 ± 0.0113	0.6145 ± 0.0577
	(0.05) - (0.6)	0.8647 ± 0.0171	0.8283 ± 0.0733	0.5178 ± 0.0569	0.9711 ± 0.0116	0.6062 ± 0.0585
	(0.05) - (0.7)	**0.8737 ± 0.0179**	**0.8309 ± 0.0953**	0.5456 ± 0.0561	0.9742 ± 0.0112	**0.6322 ± 0.0586**
	(0.1) - (0.5)	0.8654 ± 0.0156	0.8175 ± 0.0889	0.5022 ± 0.0602	0.9767 ± 0.0105	0.5911 ± 0.0556
	(0.1) - (0.6)	0.8661 ± 0.0146	0.7953 ± 0.0742	0.5033 ± 0.0617	0.9773 ± 0.0097	0.5854 ± 0.0544
	(0.1) - (0.7)	0.8651 ± 0.0179	0.8051 ± 0.0975	0.4822 ± 0.0643	**0.9827 ± 0.0087**	0.5744 ± 0.0642
DFML	(0.01) - (0.5)	**0.8984 ± 0.0153**	0.8517 ± 0.0566	**0.7033 ± 0.0646**	0.9581 ± 0.0193	**0.7350 ± 0.0497**
	(0.01) - (0.6)	0.8932 ± 0.0149	0.8526 ± 0.0614	0.6944 ± 0.0517	0.9544 ± 0.0159	0.7280 ± 0.0474
	(0.01) - (0.7)	0.8864 ± 0.0101	0.8381 ± 0.0747	0.6189 ± 0.0372	0.9683 ± 0.0127	0.6801 ± 0.0361
	(0.05) - (0.5)	0.8933 ± 0.0176	0.8429 ± 0.0782	0.7022 ± 0.0655	0.9514 ± 0.0199	0.7295 ± 0.0538
	(0.05) - (0.6)	0.8934 ± 0.0175	0.8459 ± 0.0592	0.6856 ± 0.0670	0.9570 ± 0.0165	0.7238 ± 0.0558
	(0.05) - (0.7)	0.8949 ± 0.0148	**0.8696 ± 0.0661**	0.6567 ± 0.0572	0.9679 ± 0.0102	0.7195 ± 0.0461
	(0.1) - (0.5)	0.8945 ± 0.0139	0.8431 ± 0.0848	0.6722 ± 0.0558	0.9627 ± 0.0134	0.7148 ± 0.0492
	(0.1) - (0.6)	0.8900 ± 0.0172	0.8279 ± 0.0588	0.6733 ± 0.0646	0.9565 ± 0.0168	0.7117 ± 0.0532
	(0.1) - (0.7)	0.8946 ± 0.0177	0.8691 ± 0.0840	0.6344 ± 0.0641	**0.9747 ± 0.0134**	0.7009 ± 0.0574
$DFML_{\chi^2}$	(0.01) - (0.5)	0.8887 ± 0.0241	0.7589 ± 0.0598	0.8833 ± 0.0729	0.8901 ± 0.0262	0.7884 ± 0.0549
	(0.01) - (0.6)	0.8790 ± 0.0213	0.7390 ± 0.0530	0.8733 ± 0.0567	0.8810 ± 0.0223	0.7708 ± 0.0483
	(0.01) - (0.7)	0.8888 ± 0.0180	0.7773 ± 0.0569	0.8044 ± 0.0543	**0.9146 ± 0.0220**	0.7568 ± 0.0391
	(0.05) - (0.5)	0.8806 ± 0.0185	0.7317 ± 0.0585	**0.8978 ± 0.0602**	0.8751 ± 0.0186	0.7819 ± 0.0510
	(0.05) - (0.6)	0.8859 ± 0.0181	0.7567 ± 0.0408	0.8900 ± 0.0700	0.8845 ± 0.0243	**0.7885 ± 0.0425**
	(0.05) - (0.7)	0.8918 ± 0.0194	**0.7792 ± 0.0435**	0.8389 ± 0.0615	0.9079 ± 0.0176	0.7808 ± 0.0476
	(0.1) - (0.5)	0.8856 ± 0.0223	0.7485 ± 0.0534	0.8911 ± 0.0590	0.8837 ± 0.0284	0.7879 ± 0.0492
	(0.1) - (0.6)	0.8780 ± 0.0221	0.7425 ± 0.0463	0.8722 ± 0.0495	0.8799 ± 0.0259	0.7726 ± 0.0420
	(0.1) - (0.7)	**0.8924 ± 0.0159**	0.7756 ± 0.0528	0.8422 ± 0.0596	0.9078 ± 0.0186	0.7812 ± 0.0432
$DFML_{Lap}$	(0.01) - (0.5)	**0.8687 ± 0.0194**	0.8075 ± 0.0885	**0.5256 ± 0.0729**	0.9737 ± 0.0124	**0.6060 ± 0.0706**
	(0.01) - (0.6)	0.8643 ± 0.0165	0.7908 ± 0.1072	0.5100 ± 0.0639	0.9727 ± 0.0095	0.5885 ± 0.0722
	(0.01) - (0.7)	0.8605 ± 0.0139	0.7845 ± 0.1009	0.4811 ± 0.0588	0.9768 ± 0.0117	0.5686 ± 0.0582
	(0.05) - (0.5)	0.8613 ± 0.0190	0.8091 ± 0.0913	0.4956 ± 0.0692	0.9733 ± 0.0106	0.5848 ± 0.0670
	(0.05) - (0.6)	0.8607 ± 0.0170	0.8033 ± 0.0887	0.4900 ± 0.0585	0.9744 ± 0.0112	0.5785 ± 0.0631
	(0.05) - (0.7)	0.8682 ± 0.0181	**0.8214 ± 0.0830**	0.5089 ± 0.0649	0.9783 ± 0.0100	0.6036 ± 0.0618
	(0.1) - (0.5)	0.8610 ± 0.0160	0.8028 ± 0.0827	0.4822 ± 0.0648	0.9771 ± 0.0108	0.5724 ± 0.0570
	(0.1) - (0.6)	0.8621 ± 0.0144	0.7912 ± 0.0791	0.4844 ± 0.0607	0.9779 ± 0.0091	0.5702 ± 0.0565
	(0.1) - (0.7)	0.8604 ± 0.0157	0.7901 ± 0.0913	0.4600 ± 0.0540	**0.9834 ± 0.0089**	0.5541 ± 0.0554

Al-Obeidallah, M. G., Petridis, M., Kapetanakis, S., 2018. A Multiple Phases Approach for Design Patterns Recovery Based on Structural and Method Signature Features. Int. J. Softw. Innov. 6 (3), 36—52.

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

Arcuri, A., Briand, L., 2014. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. Softw. Test. Verif. Reliab. 24 (3), 219–250.

Arevalo, G., Buchli, F., Nierstrasz, O., 2004. Detecting implicit collaboration patterns. In: Proc. 11th Working Conference on Reverse Engineering. pp. 122–131.

Bafandeh Mayvan, B., Rasoolzadegan, A., Ghavidel Yazdi, Z., 2017. The State of the Art on Design Patterns: A systematic mapping of the literature. J. Syst. Softw. 125, 93–118.

Bernardi, M. L., Cimitile, M., Di Lucca, G., 2014. Design Pattern Detection Using a DSL-driven Graph Matching Approach. J. Softw. Evol. Process 26 (12), 1233–1266.

Binun, A., Kniesel, G., 2012. Dpjf - design pattern detection with high accuracy. In: 2012 16th Eur. Conf. on Software Maintenance and Reengineering. pp. 245–254.

Chidamber, S. R., Kemerer, C. F., 1994. A Metrics Suite for Object Oriented Design. IEEE Trans. Softw. Eng. 20 (6), 476–493.

Chihada, A., Jalili, S., Hasheminejad, S. M. H., Zangooei, M. H., 2015. Source code and design conformance, design pattern detection from source code by classification approach. Appl. Soft Comput. 26, 357–367.

Cordella, L. P., Foggia, P., Sansone, C., Vento, M., 2004. A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. 26 (10), 1367–1372.

Dabain, H., Manzer, A., Tzerpos, V., 2015. Design Pattern Detection Using FINDER. In: Proc. 30th Ann. ACM Symposium on Applied Computing. pp. 1586–1593.

Di Martino, B., Esposito, A., 2016. A Rule-based Procedure for Automatic Recognition of Design Patterns in UML Diagrams. Softw. Pract. Exper. 46 (7), 983–1007.

Dong, J., Zhao, Y., Sun, Y., 2009. A Matrix-Based Approach to Recovering Design Patterns. IEEE Trans. Syst., Man, Cybern. A, Syst. Humans 39 (6), 1271–1282.

Dwivedi, A. K., Tirkey, A., Rath, S. K., 2018. Software design pattern mining using classification-based techniques. Front. Comput. Sci. 12 (5), 908–922.

Dzindolet, M. T., Peterson, S. A., Pomranky, R. A., Pierce, L. G., Beck, H. P., 2003. The role of trust in automation reliance. Int. J Hum. Comput. Stud. 58 (6), 697–718.

Eiben, A. E., Smith, J. E., 2015. Introduction to Evolutionary Computing, 2nd Edition. Springer-Verlag Berlin Heidelberg.

Ferenc, R., Beszedes, A., Fulop, L., Lele, J., 2005. Design pattern mining enhanced by machine learning. In: Proc. 21st IEEE Int. Conf. Software Maintenance (ICSM). pp. 295–304.

Fontana, F. A., Caracciolo, A., Zanoni, M., 2012. DPB: A Benchmark for Design Pattern Detection Tools. In: Proc. 16th European Conf. Software Maintenance and Reengineering (CSMR). pp. 235–244.

Fontana, F. A., 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.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gil, J. Y., Maman, I., 2005. Micro Patterns in Java Code. In: Proc. 20th Ann. ACM SIGPLAN Conf. Object-oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 97–116.

Grosan, C., Abraham, A., 2011. Rule-Based Expert Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 149–185.

Guéhéneuc, Y.-G., 2007. P-mart: Pattern-like micro architecture repository. Proc. 1st EuroPLoP Focus Group on Pattern Repositories.

Guéhéneuc, Y.-G., Antoniol, G., 2008. DeMIMA: A multilayered approach for design pattern identification. IEEE Trans. Softw. Eng. 34 (5), 667–684.

Published version available at: https://doi.org/10.1016/j.jss.2021.110919

Guéhéneuc, Y.-G., Guyomarc'H, J.-Y., Sahraoui, H., 2010. Improving Design-pattern Identification: A New Approach and an Exploratory Study. Software Qual. J. 18 (1), 145–174.

Gueheneuc, Y.-G., Sahraoui, H., Zaidi, F., 2004. Fingerprinting design patterns. In: Proc. 11th Working Conf. Reverse Engineering. pp. 172–181.

Hadi, W., Aburub, F., Alhawari, S., 2016. A new fast associative classification algorithm for detecting phishing websites. Appl. Soft Comput. 48, 729–734.

Hayashi, S., Katada, J., Sakamoto, R., Kobayashi, T., Saeki, M., 2008. Design pattern detection by using meta patterns. IEICE Trans. Inf. Syst. E91.D (4), 933–944.

Kotsiantis, S., Zaharakis, I., Pintelas, P., 2006. Machine learning: a review of classification and combining techniques. Artif. Intell. Rev. 26, 159–190.

Koza, J. R., 1992. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press.

Kramer, C., Prechelt, L., 1996. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In: Proc. 3rd Working Conference on Reverse Engineering (WCRE). pp. 208–215.

Li, W., Han, J., Pei, J., 2001. CMAR: Accurate and Efficient Classification Based on Multiple Class-Association Rules. In: Proc. 2001 IEEE Int. Conf. Data Mining (ICDM). pp. 369–376.

Liu, B., Hsu, W., Ma, Y., 1998. Integrating classification and association rule mining. In: Proc. 4th Int. Conf. Knowledge Discovery and Data Mining (KDD). pp. 80–86.

Lucia, A. D., Deufemia, V., Gravino, C., Risi, M., 2009. Design pattern recovery through visual language parsing and source code analysis. J Syst. Softw. 82 (7), 1177–1193.

Lucia, A. D., Deufemia, V., Gravino, C., Risi, M., 2018. Detecting the Behavior of Design Patterns Through Model Checking and Dynamic Analysis. ACM Trans. Softw. Eng. Methodol. 26 (4), 13:1–13:41.

Luna, J. M., Romero, J. R., Ventura, S., 2012. Design and Behavior Study of a Grammar-guided Genetic Programming Algorithm for Mining Association Rules. Knowl. Inf. Syst. 32 (1), 53–76.

Mayvan, B. B., Rasoolzadegan, A., 2017. Design pattern detection based on the graph theory. Knowl.-Based Syst. 120, 211–225.

Mori, T., Uchihira, N., 2019. Balancing the trade-off between accuracy and interpretability in software defect prediction. Empir. Software Eng. 24, 779–825.

Ng, J. K.-Y., Guéhéneuc, Y.-G., Antoniol, G., 2010. Identification of Behavioural and Creational Design Motifs Through Dynamic Analysis. J. Softw. Maint. Evol. 22 (8), 597–627.

Niere, J., Schäfer, W., Wadsack, J. P., Wendehals, L., Welsh, J., 2002. Towards Pattern-based Design Recovery. In: Proc. 24th Int. Conf. Software Engineering (ICSE). pp. 338–348.

Poelmans, J., Ignatov, D. I., Kuznetsov, S. O., Dedene, G., 2013. Formal Concept Analysis in Knowledge Processing: A Survey on Applications. Expert Syst. Appl. 40 (16), 6538–6560.

Pree, W., Sikora, H., 1997. Design patterns for object-oriented software development (tutorial). In: Proc. 19th Int. Conf. Software Engineering (ICSE). pp. 663–664.

Rana, R., Staron, M., Berger, C., Hansson, J., Nilsson, M., Meding, W., 2014. The Adoption of Machine Learning Techniques for Software Defect Prediction: An Initial Industrial Validation. In: Proc. Joint Conference on Knowledge-Based Software Engineering (JCKBSE). pp. 270–285.

Rasool, G., Mader, P., 2011. Flexible Design Pattern Detection Based on Feature Types. In: Proc. 26th IEEE/ACM Int. Conf. Automated Software Engineering. pp. 243–252.

Rasool, G., Philippow, I., Mäder, P., 2010. Design Pattern Recovery Based on Annotations. Adv. Eng. Softw. 41 (4), 519–526.

Shi, N., Olsson, R. A., 2006. Reverse Engineering of Design Patterns from Java Source Code. In: Proc. 21st IEEE/ACM Int. Conf. Automated Software Engineering. pp. 123–134.

Smith, J. M., 2012. Elemental Design Patterns, 1st Edition. Addison-Wesley Professional.

Thabtah, F., 2007. A Review of Associative Classification Mining. Knowl. Eng. Rev. 22 (1), 37–65.

Thaller, H., Linsbauer, L., Egyed, A., 2019. Feature Maps: A Comprehensible Software Representation for Design Pattern Detection. In: Proc. IEEE 26th Int. Conf. Software Analysis, Evolution, and Reengineering (SANER). pp. 207–217.

Tonella, P., Antoniol, G., 2001. Inference of Object-oriented Design Patterns. Journal of Software Maintenance 13 (5), 309–330.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S. T., 2006. Design Pattern Detection Using Similarity Scoring. IEEE Trans. Softw. Eng. 32 (11), 896–909.

Uchiyama, S., Washizaki, H., Fukazawa, Y., Kubo, A., 2011. Design pattern detection using software metrics and machine learning. In: Proc. 1st Int. Workshop Model-Driven Software Migration (MDSM). p. 38.

Ventura, S., Romero, C., Zafra, A., Delgado, J. A., Hervás, C., 2008. JCLEC: a java framework for evolutionary computation. Soft Comput. 12 (4), 381–392.

Wierda, A., Dortmans, E., Somers, L., 2009. Pattern Detection in Object-Oriented Source Code. In: Proc. Int. Conf. Software and Data Technologies. pp. 141–158.

Xiong, R., Li, B., 2019. Accurate Design Pattern Detection Based on Idiomatic Implementation Matching in Java Language Context. In: IEEE 26th Int. Conf. Software Analysis, Evolution and Reengineering (SANER). pp. 163–174.

Yin, X., Han, J., 2003. CPAR: Classification Based on Predictive Association Rules. In: Proc. SIAM Int. Conf. Data Mining. pp. 331–335.

Yu, D., Zhang, P., Yang, J., Chen, Z., Liu, C., Chen, J., 2018. Efficiently detecting structural design pattern instances based on ordered sequences. J Syst. Softw. 142, 35–56.

Yu, D., Zhang, Y., Chen, Z., 2015. A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. J Syst. Softw. 103, 1–16.

Zanoni, M., Fontana, F. A., Stella, F., 2015. On applying machine learning techniques for design pattern detection. J. Syst. Softw. 103, 102–117.

Published version available at: https://doi.org/10.1016/j.jss.2021.110919