Feature Maps: A Comprehensible Software Representation for Design Pattern Detection

Hannes Thaller, Lukas Linsbauer, and Alexander Egyed
Institute for Software Systems Engineering
Johannes Kepler University Linz, Austria
{hannes.thaller, lukas.linsbauer, alexander.egyed}@jku.at

***Abstract*—Design patterns are elegant and well-tested solutions to recurrent software development problems. They are the result of software developers dealing with problems that frequently occur, solving them in the same or a slightly adapted way. A pattern's semantics provide the intent, motivation, and applicability, describing *what* it does, *why* it is needed, and *where* it is useful. Consequently, design patterns encode a well of information. Developers weave this information into their systems whenever they use design patterns to solve problems. This work presents *Feature Maps*, a flexible human- and machine-comprehensible software representation based on micro-structures. Our algorithm, the *Feature-Role Normalization*, presses the high-dimensional, inhomogeneous vector space of micro-structures into a feature map. We apply these concepts to the problem of detecting instances of design patterns in source code. We evaluate our methodology on four design patterns, a wide range of balanced and imbalanced labeled training data, and compare classical machine learning (Random Forests) with modern deep learning approaches (Convolutional Neural Networks). Feature maps yield robust classifiers even under challenging settings of strongly imbalanced data distributions without sacrificing human comprehensibility. Results suggest that feature maps are an excellent addition in the software analysis toolbox that can reveal useful information hidden in the source code.**

***Index Terms*—feature maps, micro-structures, design patterns, machine learning, random forest, deep learning, convolutional neural networks, program comprehension, reverse engineering**

I. Introduction

Design Patterns (DPs) are elegant and well-tested solutions to recurrent software development problems. *Design Patterns – Elements of Reusable Object-Oriented Software*, by Gamma et al. [1], is the best-known collection of patterns and inspiration for many follow-ups. They are the result of software developers dealing with problems that frequently occur, solving them in the same or a slightly adapted way. DPs are the generalization of the different adapted implementations, such that they can be reused and applied over and over again in different situations. They solve high-level Object-Oriented (OO) architectural problems dealing with creation, structure or behavior of a small set of classes or objects, but may also circumvent deficiencies and inflexibilities in OO languages.

Pattern descriptions are very detailed and contain their name, intent, motivation, where they are applicable, structures, participants, collaborations, and so forth [1]. A pattern's semantic is given by the intent, motivation, and applicability, which describe *what* the pattern does, *why* the pattern is needed, and *where* it is useful. Developers weave this information encoded as design patterns into their system as they use them to solve problems. The usage of a pattern is related to specific design decisions during development. However, fast development cycles often prohibit the documentation of these decisions and their rationales. Similarly, the actual usage of the pattern is seldom documented. Hence decision, rationale and their materialization in the form of the pattern's implementation are lost in the system's source code. Retrieving this encoded information such that development, redevelopment, and maintenance can profit from it is the primary motivation of Design Pattern Detection (DPD). For the sake of simplicity, we will use the term design pattern detection to describe the process of detecting *instances* of design patterns. DPD is especially useful for preliminary analysis in maintenance and testing scenarios where the code is unknown or undocumented. The detected patterns hint at structures and dependencies, highlight algorithms and their moving parts, and help to find performance-critical regions.

The biggest challenge in DPD is that patterns are only a guideline for implementing a specific solution; hence each pattern can be implemented in various ways. Each pattern implementation variant resembles the original intent of the pattern but may diverge drastically if compared to other implementation variants. Detecting all variants and mapping the classes to pattern roles is a non-trivial task as enumerating the different variants is not sufficient for real-world setups and their inherent variations.

We use Feature Maps (FMs) as input to Random Forests (RFs) [2] and Convolutional Neural Networks (CNNs) [3] to decide whether a given set of classes maps to the roles of a specific design pattern. These feature maps are human-interpretable, stacked, named subtrees (micro-structures) extracted from a system's Abstract Semantics Graph (ASG) that can be used as a software representation for a small set of classes. Results indicate that feature maps are an excellent approach to represent software, enabling robust DPD even if DP instances are highly under-represented. More specifically, the contributions of this work are:

- a new, flexible and comprehensible software representation called *feature maps* that are useful for software analysis,
- an approach for *detecting instances of design patterns in source code* by using *feature maps* in conjunction with *supervised machine learning*,

- an evaluation and methodology proposal of reproducible and comparable design pattern detection.

Section II provides background to design patterns and machine learning. Section III describes our approach for detecting design pattern instances with feature maps. Section IV presents the experiment setup and evaluation results. A general discussion is given in Section V and threats to validity are discussed in Section VI. Section VII summarizes related research and Section VIII concludes this work and offers prospects of design pattern detection with feature maps and machine learning.

II. Background

Two ingredients are crucial for the approach within this work. First, Design Patterns (DPs) and their intricacies from a Design Pattern (Instance) Detection (DPD) perspective influence how detectors are built and evaluated. Thus a more formal definition helps to frame the problem and its solution. Secondly, Machine Learning (ML) and its automation capabilities concerning data modeling are essential to understand the approach and its evaluation.

A. Design Patterns

A Design Pattern (DP) is a set of roles to which participating classes are mapped. These classes communicate (structure and collaboration) in an organized fashion and have specific semantics (intent, motivation, and applicability) attached to them. Design pattern (instance) detection reconstructs the original mappings between classes of a system and roles of a pattern with respect to their communication such that the attached semantics provide information about the system under inspection.

Figure 1 (based on an example by Gamma et al. [1]) illustrates this process in which the input is a set of classes, and the output is a set of mappings between these classes and the roles of a specific pattern. In other words, the task is to annotate a set of classes with the roles of a design pattern. The example is a subsystem of a drawing tool that uses the Composite pattern to interact, in a uniform way, with the drawing primitives and the scene. The Composite pattern organizes objects into a tree structure where internal nodes delegate specific calls while leaf nodes implement the actual behavior of the call. `Graphic` is the superclass of `Picture`, `Line`, and `Rectangle` and defines a common interface for these. `Picture` aggregates `Graphic` objects, and delegates calls to the *draw*-method. `Line` and `Rectangle` provide the actual implementation of *draw*, hence are called leafs. The goal is now to find a process that reconstructs the original mappings between classes and pattern roles indicated by the question mark in Figure 1. The left side in Figure 1 represents the initial state of the system before the detection process. The right side shows the system with annotated roles after the detection process. Mapping 1, for example, maps `Graphic` $\mapsto$ *Component*, `Picture` $\mapsto$ *Composite* and `Line` $\mapsto$ *Leaf*.

Each role mapping assigns at least one class to one role resulting in multiple possible role mappings for the same subsystem given various implementations of the different roles. Mappings 1 and 2 differ only in one role mapping which is a common scheme in design patterns. *Primary roles* define the communication scheme within the pattern and drive the communication through a pattern's class structure, hence are often abstractions. *Secondary roles* provide the implementation for the abstractions and inherit the protocol from the primary roles, thus are commonly fluctuating in their class assignment. A system usually provides multiple versions of the secondary roles but only a handful of different implementations for the primary roles. All mappings that share the same primary roles belong to the same *unique role mapping* representing one specific implementation of a pattern within a subsystem. More formally, a pattern mapping is a k-fold relation between a set of classes $\boldsymbol{C}$ and a set of roles $\boldsymbol{R}$ with

$$m_{P^k} = \{(\boldsymbol{a}, \boldsymbol{b}) \in \boldsymbol{C}^k \times \boldsymbol{R}^k : \boldsymbol{a} \text{ complies with } \boldsymbol{b}\}, \qquad (1)$$

in which P^k is a specific pattern with k roles. Each unique role mapping reflects an equivalence class in which the *primary roles* are compared. Given a set of pattern mappings $\boldsymbol{M}_{P^k}$ with the equivalence relation $\sim_{P^k}: M_{P^k} \times M_{P^k}$ in which the classes mapped to the primary roles are compared, then

$$\langle m \rangle_{\sim_{P^k}} = \{x \in \boldsymbol{M}_{P^k} : x \sim_{P^k} m\} \qquad (2)$$

represents the P^k equivalence class from m. That is, all role mappings of pattern P with k roles that have the same classes $\boldsymbol{C}$ mapped to the primary roles $\boldsymbol{R}$. In Figure 1, `Graphic` maps to the *Component* role representing the primary role of the Composite pattern. `Line`, `Rectangle`, and `Picture` map to the secondary roles where Mappings 1-2 belong to the *Unique Mapping A*. Tertiary roles (e.g., Client) that solely function as an entry point for the pattern's communication sequence are often ignored in detection processes as they do not carry any useful information. The concept of unique mapping carries importance in analysis settings where they explicitly state the boundaries for design pattern instances.

B. Machine Learning

Machine Learning (ML) describes methods that learn relationships or structure from data in an automated fashion. Essential elements of ML are data (e.g., images or time series), the model (e.g., Random Forest [2] or CNN [3]), the optimization procedure (e.g., Adam [4]) and the evaluation procedure (e.g., Cross-Validation [5] or test-set [6] method).

1) Data: The DPD problem can be framed as a supervised classification problem in which observations are annotated with the ground truth (e.g., is an instance of Composite, is not an instance of Composite). ML algorithms often expect the data to be independent and identically distributed (i.i.d.) meaning that the observations are mutually independent and collected in the same fashion. Especially observation independence is vital as violating the assumption results over-optimistic model performance in classification settings.

Data can be preprocessed by standardizing or rescaling the values into a specific range. In addition to normalizing, data permutation is often employed to increase the amount of

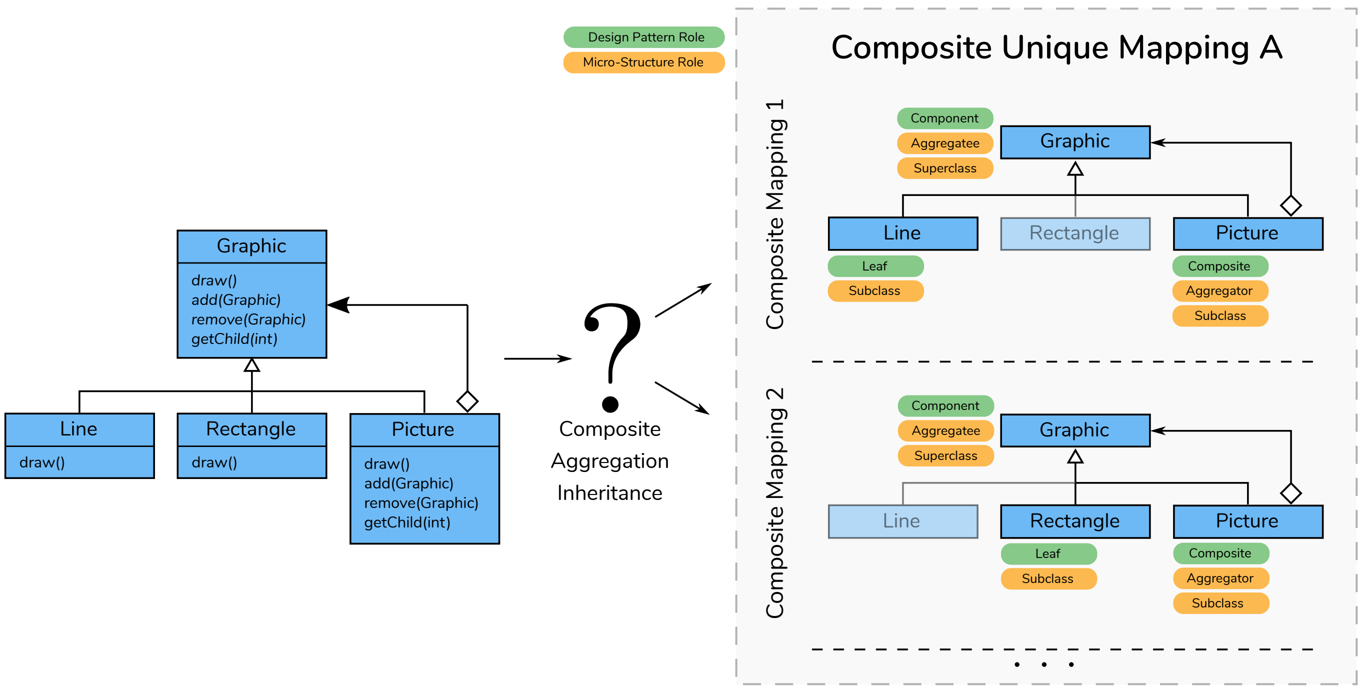

Figure 1. The goal is to find a process that can reconstruct the original role mappings of a given pattern. Inputs are classes of a system, outputs are mappings between pattern roles and the input classes. A unique mapping defines a set of mappings that share the same classes mapped to the primary roles. Component (Graphic) is assigned to the primary role and defines the communication scheme, while *Composite* (Picture), *Leaf* (Line and Rectangle) are classes assigned to secondary roles. In addition to the Composite design pattern, the example also presents the detection of the Aggregation and Inheritance micro-structures. Similar to the detection of design patterns, micro-structures are only an assortment of classes that form a class structure. The Composite example is based on an example by Gamma et al. [1].

available data or to fit models that are robust against simple transformations (i.e., reduce overfitting) [7]. For instance, rotating or mirroring an image does not change the objects it shows (e.g., rotated apple is still an apple).

2) Model: Many different supervised machine learning models of varying complexity and with different strengths and weaknesses exist.

Convolutional Neural Networks (CNNs) [3] are fundamental building blocks in modern machine learning and are prominent for their capabilities for computer vision problems. They learn local correlations within volumes and combine these correlations into high-level features at later stages. For instance, nearby pixels often correlate with each other, e.g., color and texture of an apple on the top right corner of an image. CNNs innately model such local correlations leading to good results in many domains while still having a reasonable amount of model parameters.

A *Decision Tree* [8] recursively partitions the input space via axis-parallel splits that can be represented as a tree. Each leaf carries a response (e.g., a pattern class) while the nodes represent boolean conditions on an input dimension (e.g., is an interface $\mapsto$ {yes, no}). A *Random Forest (RF)* [2] contains multiple randomly perturbed decision trees, that reduce the possible high variance of a single tree. The trees are perturbed by fitting each on a subset of the data which creates smoother decision boundaries as multiple splits are averaged in the final prediction.

Hyper-parameter optimization searches the space of possible parameter configurations of an ML algorithm that leads to the best performing model according to some predefined goal function. For instance, *tree depth* or *partition criterion* are hyper-parameters of a decision tree.

3) Evaluation: It is crucial that the evaluation procedure provides a measurement of the model that is truthful, i.e., it should not over- or underestimate the performance. Cross-Validation [9] (CV) is a method to evaluate the generalization performance by splitting the dataset into k-folds. $k-1$ folds are used to fit a model while the remaining fold is used to estimate its generalization performance. This process is repeated k times leading to k models and estimates that are averaged into a global generalization performance of the ML algorithm regarding the dataset and parameters. CV is a nearly unbiased estimator for the generalization performance except in the case of small datasets in which the evaluation variance may misestimate the generalization. Repeated cross-validation may be used to reduce the variance, trading it for some bias.

Accuracy, Precision, Recall or the Matthews Correlation Coefficient (MCC) [10], [11] are performance metrics for classification models. Accuracy, Recall, and Precision are frequently used metrics and need no further explanation. MCC is a reliable and balanced performance metric for binary classification ranging from -1 to 1 describing the strength of association between model prediction and ground truth. It provides the most accurate measurement of a (binary) model's performance even if the dataset is imbalanced (skewed distribution of prediction labels). This does not hold for Accuracy, Precision and Recall as they do not consider all types of correctly or incorrectly predicted instances (true/false positive, true/false negatives).

III. Design Pattern Instance Detection Pipeline

Figure 2 shows a typical multistage DPD pipeline similar to many DP detectors [12], [13], [14], [15], [16], [17]. Rectangles are artifacts while ellipses are processes. *Source Code* and *Probability* are the input and output of the entire DPD pipeline while *Micro-structures*, *Candidate Role Mappings* and *Feature Maps* are intermediate artifacts. *Feature Extraction* extracts distinctive features from the source code. *Candidate Sampling* reduces the total number of mappings that need to be evaluated. *Feature Normalization* transforms the features into a homogeneous space such that the *Design Pattern Inference* can reason efficiently about it.

A. Feature Extraction

Feature extraction deals with the inherent complexity of programming languages by extracting high-level concepts that later stages can use to successfully find pattern instances (see Figure 2). DPs describe classes and their loose arrangements and communication paths. Consequently, extracted features ideally capture these arrangements and their relationships to improve reasoning in later stages.

We use Micro-Structures (MSs) [18] as features, which are very small DPs (usually one or two roles). A MS describes a general structural or behavioral property within or between a set of classes (e.g., self-reference, inheritance or method calls). Casting MSs into an easily readable form allows developers and algorithms to effectively comprehend these properties. This includes fast discovery of inheritance and call dependencies but also complex aspects like the implementation of a Composite or Decorator pattern.

The most outstanding difference between MSs and DPs is that the former can be detected using logic or pattern matching, i.e., their size prohibits variance in their actual implementation. MS detectors are sub-graph filters retaining only sub-graphs that match their specified predicate. For example, $Inheritance(sup, sub) : \mathbb{T}^2, (sup, sub) \mapsto Ancestor(sup, sub)$, describes the logic to filter for the *Inheritance* MS. Given two Type nodes, sup has to be an ancestor of sub in order to fulfill the $Inheritance$ predicate. The result of such an MS detector are sub-graphs from the program's ASG that are annotated with the MS roles illustrated in Figure 1 for the *Inheritance* and *Aggregation* MS. `Graphic` $\mapsto$ *Superclass* and `Picture` $\mapsto$ *Subclass* for one possible *Inheritance* MS instance. A complete catalog of Micro-Structures and their detectors is given in our previous work [19], which is a refined and extended catalog based on Maggioni's work [20]. The catalog is made up of three sub-catalogs: Elemental Design Patterns [21], Design Pattern Clues [22] and Micro-Patterns [23]. Each sub-catalog was defined independently with different motivations and goals, but all of them prove valuable in the process of DPD as Arcelli Fontana et al. [24] concluded in a series of experiments.

B. Candidate Sampling

Candidate Sampling uses the extracted features (and sometimes the *Source Code*) to find potential candidates of design pattern instances in a program's ASG. As discussed, role mappings link concrete classes to specific pattern roles which span a search space of $\binom{n}{k}$ potential mappings where n is the number of available classes and k the number of roles that need to be mapped for a pattern. Hence, finding potential candidates is vital since a full-search is impossible for non-trivial systems. We used Heuristic Search [19] that we proposed in our previous work as candidate sampler. Heuristic Search checks for graph isomorphism [25] between a very general description of the pattern and classes in the program under inspection. It uses primary roles as an entry point to iteratively search for secondary roles that fit the pattern description. Heuristic Search reduces the usual $2^{O(\sqrt{n \log(n)})}$ isomorphism search [26] to a search that is linear with the number of types. For Composite, heuristic search executes the following steps: 1) Collect all types $\boldsymbol{sup}$ that are super-types as *Component* ($\boldsymbol{sup} \mapsto$ *Component*); 2) Collect all types $\boldsymbol{sub}$ that are sub-types from $\boldsymbol{sup}$ and aggregate $\boldsymbol{sub}$ as *Composite* ($\boldsymbol{sub} \mapsto$ *Composite*); 3) Collect all types $\boldsymbol{sib}$ that are sub-types from $\boldsymbol{sup}$ and do not aggregate $\boldsymbol{sup}$ as *Leaf* ($\boldsymbol{sib} \mapsto$ *Leaf*). Each found role mapping is a candidate that is later checked in the design pattern inference stage for their correct consistency with the pattern. A drawback to this heuristic search is that it might miss potential pattern instances. The advantage is that these are simple and efficient algorithms to find potential candidates.

C. Feature Normalization

Feature-Role Normalization (FRN) is the main contribution of this work. It is used to normalize micro-structures into a fixed sized matrix called Feature Map (FM). FRN allows the usage of a wide variety of existing statistical methods while still retaining the interpretability for manual analysis via software engineers. Figure 2 shows the pipeline in which first the features (micro-structures) are extracted followed by the sampling of candidates from the system. A candidate mapping is a set of classes that map to the design pattern roles which may, may not or may partially map to several micro-structures. Each of the potentially mapped micro-structures may have a different number of roles, resulting in an inhomogeneous feature space. This inhomogeneous feature space is problematic in the inference step as most machine learning methods cannot handle irregularly structured data. Feature-Role Normalization presses this inhomogeneous feature space into a homogeneous two-dimensional feature space that is comprehensible and maintains enough information to be useful for engineers and learning algorithms. More specific, the goal of the normalization, is to take $n = |\boldsymbol{f}|$ features each having $m_{fi} = |\boldsymbol{r}_{fi}|$ roles and map them to $k = |\boldsymbol{r}_{di}|$ design pattern roles such that the result is a fixed-sized matrix. FRN provides one approach to this problem by creating an $n \times k$ matrix where rows represent

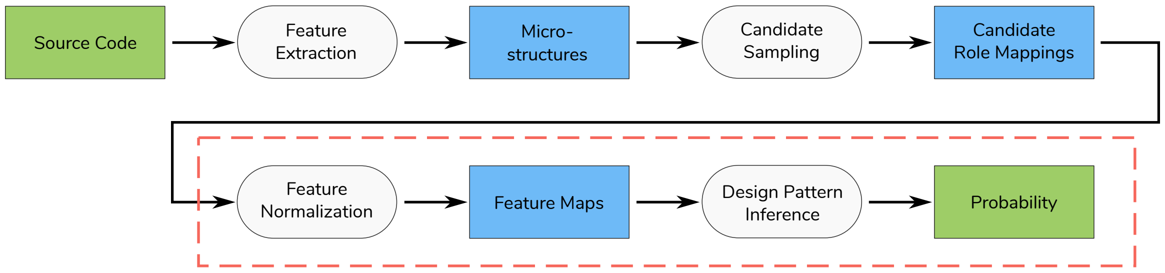

Figure 2. Overview of the detection pipeline. Ellipses are processes; rectangles are artifacts. First, class features are extracted from the source code. Then, based on the features, pattern instance candidates are sampled from the system. Optionally, the features are normalized and ultimately classified by the inference method.

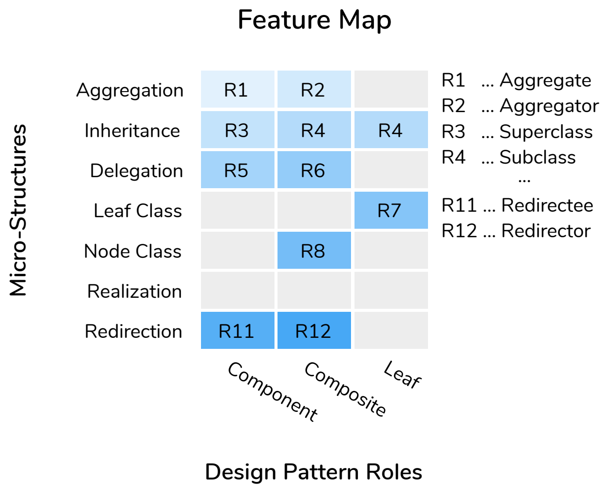

Figure 3. A (condensed) feature map is a stack of micro-structures for a set of classes (represented by their Decorator roles in this case). Values represent unique role identifiers with $id \in \mathbb{N}$ or any other property related to micro-structures (e.g., number of occurrences of the micro-structure). The coloring in this example is a gradient based on the role identifiers. Gray spots represent absent values (zeros).

micro-structures and columns design pattern roles. Values in the feature map, in the context of DPD, are globally unique role identifiers ($id \in \mathbb{N}$), where no two roles of any pattern have the same values associated. That said, roles might be shared across multiple features if they share similar semantic information. Zeros represent missing features meaning that the class mapped to the role does not participate in the micro-structure.

Figure 3 illustrates an example of a feature map describing classes that make up a Composite pattern. Values are role identifiers colored by a simple gradient, and empty gray spots are absent mappings (zeros). Rows depict an illustrative subset of the micro-structures (features) and columns the design pattern roles to which a set of classes are mapped. The class assigned to *Component* is a *Superclass* with its children classes assigned to *Composite* and *Leaf*. *Composite* aggregates a *Component*, delegating and redirecting method calls to it.

This short example mirrors how software engineers would, in a slow fashion, go through the source code of a project to find possible instances of a Composite pattern. Feature maps simplify this process by compacting the relationships between multiple classes into new high-level concepts (e.g., a design pattern) and offer a straightforward way of visualizing them. Naturally, the visualization can be extended by recoloring, grouping or filtering the entries of a feature map, or by visualizing another source code property than identifiers (e.g., number of invocations between roles or complexity measures). This makes FMs a strong visualization technique for software analysis. They provide a compact view of specific aspects of a small set of classes allowing a quick overview of their relationships and class characteristics. Furthermore, the matrix structure has clear advantages regarding automation as many methods, e.g., machine learning algorithms, are designed to work with flat data.

Detection processes that operate on structured data may circumvent the use of FMs for DPD by directly operating on the ASG. Examples are Support Vector Machines [27] (with tree-kernels), Recursive Neural Networks [28], sequence-to-sequence learning models like Long Short-Term Memory Networks [29] or Graph Convolutional Neural Networks [30]. These learning algorithms and their respective models might provide possible performance advantages in the context of DPD by naturally handling the ASG and the MSs. However, some lack computational efficiency while others lack interpretability of the classification decisions.

1) Information Preservation Issues in Feature Maps: FRN normalizes the inhomogeneous feature space caused by micro-structures and their roles into a fixed-sized feature map. The resulting feature map is a sparse representation of attributes and relationships within and between nodes in an ASG. While inspired by adjacency matrices, feature maps are not a proper and full representation of the subgraphs; hence they suffer from two specific issues: noise and collisions.

A feature introduces *noise* if it has at least two roles and only one of its roles map to a class within the FM. Figure 4 illustrates this problem with the Composite pattern and the features Pure Type, Inheritance and Delegation. On the left side, the class assigned to *Leaf* is a *Superclass*. However, neither of the other Composite roles participate in this inheritance hierarchy (both are 0). The *Superclass* role of the *Leaf* class exists in the project but does not carry any meaningful information in the context of the current FM mapping. These out of context values inflate the feature map with irrelevant information and introduce noise reducing the usability of FMs. A simple solution to the

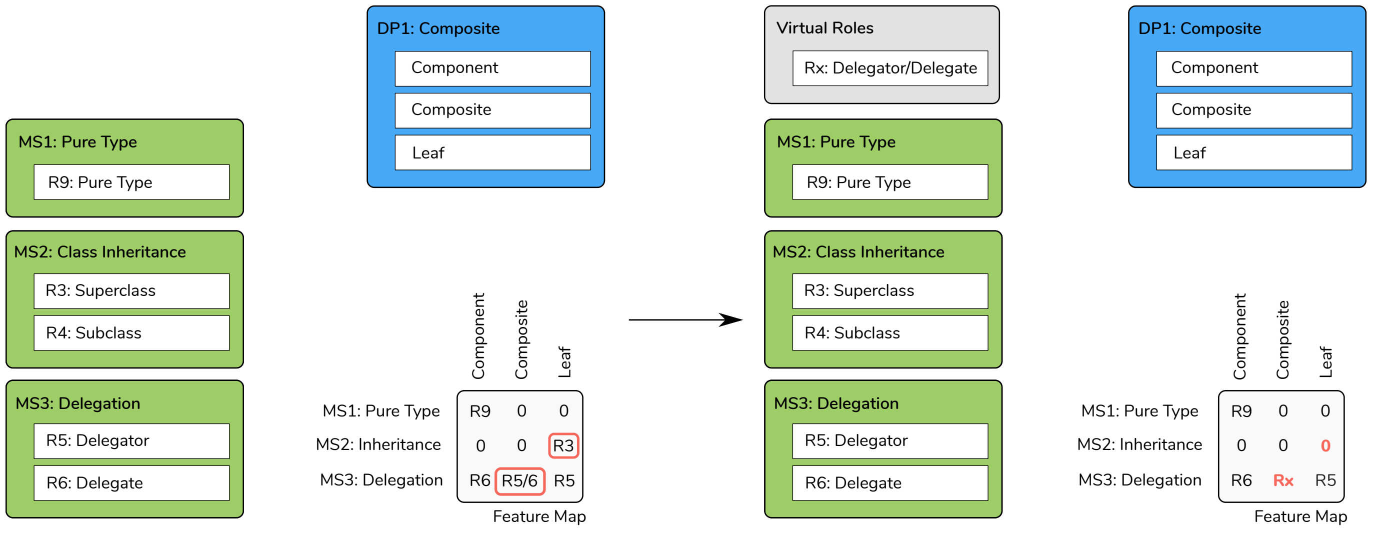

Figure 4. The left side illustrates an example of noise (R3) and role collisions (R5/6) within feature maps. Noise is introduced by features that relate to classes that do not participate in the actual mapping. Role collisions occur if a feature is present multiple times with the same set of mapped classes. The right side provides a solution to both issues where mappings to classes that do not participate in the actual mapping are ignored (forced to 0), and role collisions are solved by introducing virtual roles (Rx) that represent multiple roles at the same time.

problem is to force entries to zero in cases where features depict a relationship, but none of the other mapped classes participate in it. Figure 4 depicts this solution on the right side where *R3* becomes 0.

A feature suffers from a role *collision* if it has at least two roles and multiple instances of the feature trace to the same classes. Figure 4 (left) illustrate a collision where two instances of the Delegation MS are present in the classes assigned to *Component*, *Composite*, and *Leaf*. *Composite* delegates to *Component* and *Leaf* delegates to *Composite*. The multi-assignment for *Composite* is the result of the class-level abstraction and compactness of FMs. One possible solution is to introduce virtual roles that describe two or more roles like *Rx* in Figure 4 on the right. Virtual roles incur a certain amount of information loss (direction of relationship) especially if multiple virtual roles have to be used for a feature in a FM. However, virtual roles also allow to express self-loops, for instance if a class aggregates itself (it is *Aggregator* and *Aggregate* at the same time). These self-loops are helpful in many cases, for example, 50% of the Singleton pattern instances contain *Creation Site / Type To Create* indicating that the class is the factory for itself. This information is essential in the context of Singleton detection.

D. Design Pattern Inference

Design Pattern Inference is the last step of the DPD pipeline (Figure 2) and receives feature maps as input. During training time the models and their parameters are optimized toward a labeled dataset of feature maps. During production time the models only return the probability that an FM belongs to the model's optimized pattern or not (no labels, no model parameter modifications). The detailed intricacies of the training and production time are subject to the selected model class and may differ quite drastically. For instance, a CNN is optimized via a gradient-based algorithm while decision trees are constructed by splitting the input dimension values via an information theoretical criterion (training time).

IV. Design Pattern Instance Detection Study

This study frames the process of DPD (see Figure 1) as a binary classification problem. Each design pattern is detected by a separate model that provides the probability that the input FM belongs to the design pattern. Naturally, multi-label classification, i.e., multiple patterns per observation (FM), is supported by feeding the same FM into different models.

Given this setup, we conducted several experiments with FMs in the context of design pattern instance detection. For better reproducibility and comparability we evaluated only the last two stages of the DPD pipeline highlighted in Figure 2. Experience shows that reimplementing the entire pipeline of a DPD approach including the intermediates (e.g., sampler results) while making the detection results comparable is nearly impossible. That said, the entire pipeline from Figure 2 was implemented, and details about the specific implementation of micro-structures and candidate samplers are given in our previous work [19]. The following evaluation and experimental methodology should be seen as a minimal effort to guarantee comparable research in the future.

A. Controlled Variables

Detecting design pattern instances can be a daunting task as there are numerous ways to encode software (e.g., feature maps), select potential candidates (candidate sampling), and build classification models (e.g., RFs or CNNs). In total, the study controls for 7 Experiment Parameters (ExP).

1) **Pattern:** Four widely used patterns are considered {*Singleton, Template Method, Composite, Decorator*}. This selection is based on previous studies [17], [31], [24],

[15] but also represents an even selection regarding their categorical distribution (creational, structural and behavioral patterns).

2) **Role Count:** A confounding variable is the number of roles of the pattern. The expectation is that patterns with one role are easier to detect than patterns with 2 or 3 roles. However, more roles also imply more conditions through which better precision in the classification can be achieved. The selected patterns have a linear role count distribution with {*1, 2, 3, 4*} roles for {*Singleton, Template Method, Composite, Decorator*} excluding tertiary roles.
3) **Classification Model:** {*Random Forest, CNN*} were selected as classification model classes. The rationale for selecting CNNs is their natural ability to handle volumes, i.e., to process FMs while leveraging their structural information. RFs were selected as they are a popular and efficient choice in many ML tasks and are more light-weight regarding hyper-parameters in contrast to CNNs.
4) **Negative-Positive Candidate Ratio:** Non-trivial software systems expose a combinatorial number of possible class mappings that can function as pattern candidates. This means a practically infinite number of pattern counterexamples can be found within a system. The candidate samplers (Section III-B) that mitigate this issue favor *recall* leading to imbalanced distributions of positive and negative examples. Negative-Positive Candidate Ratio (NPCR) captures this skew in the data. The study explores NPCRs of {*1, 2, 4, 6, 8, 10*}.
5) **Data Augmentation:** Shuffling of rows was used as data augmentation method to increase the dataset size and classification difficulty. This inhibits the classifier from learning specific relationships between rows, instead of their content. A permutation count of {*0, 1, 5, 10*} states how often the dataset was copied, rows augmented and added to the dataset. Permuting rows *and* columns was not considered as it would result in a nearly random matrix destroying any structural information captured in feature maps.
6) **Optimization Budget:** A budget of {*200*} evaluations was available for each model class (RF or CNN). These evaluations were used to tune their hyper-parameters (such as the number of units within layers, or depth of tree). The optimization selected hyper-parameter configurations that maximized the MCC.
7) **Instance Independence:** Describes the confounding variable that instances within a unique mapping correlate thus result in too optimistic classification results if not properly handled. Using standard cross-validation mixes mappings from the same unique mapping into training and test-folds making the folds not independent. This leads to too optimistic evaluation results as the training examples *leak* information between training and test phase. We used {*project-fold cross validation*} to avoid information leaks (correlations between samples) during the trials.

Table I
THE 9 PROJECTS CONTAINED IN THE P-MARt DATASET. FOR EVERY PROJECT THE ORIGINAL AND THE REVISED (EXCLUDING INSTANCES OUTSIDE OF THE AVAILABLE SOURCE CODE) NUMBER OF DESIGN PATTERN INSTANCES (I.E. MAPPINGS) AS WELL AS THE NUMBER OF UNIQUE MAPPINGS IS GIVEN.

Project	**Original / Revised / Unique**			
	Singleton	Template Method	Composite	Decorator
JHotDraw	2 / 2 / 2	68 / 34 / 2	3840 / 960 / 1	176 / 44 / 1
JRefactory	2 / 2 / 2	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
JUnit	2 / 2 / 2	0 / 0 / 0	222 / 74 / 1	396 / 132 / 1
Lexi	2 / 2 / 2	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
MapperXML	3 / 3 / 3	88 / 44 /4	340 / 85 / 1	0 / 0 / 0
Netbeans	- / - / -	- / - / -	- / - / -	- / - / -
Nutch	1 / 1 / 1	14 / 7 / 2	0 / 0 / 0	0 / 0 / 0
PMD	0 / 0 / 0	1 / 0 / 0	3 / 0 / 0	0 / 0 / 0
QuickUML	1 / 1 / 1	0 / 0 / 0	120 / 30 / 2	0 / 0 / 0
Total	**13 / 13 / 13**	**171 / 85 / 8**	**4525 / 1149 / 5**	**572 / 176 / 2**

B. Response Variables

The result of each experiment trial was the generalization performance of the classifier evaluated through cross-validation and measured by {*Accuracy, Precision, Recall, F1, Matthews Correlation Coefficient*}. MCC was selected as the primary evaluation metric as it provides the most unbiased single number metric for binary classification.

C. Data Source

A total of 4 widely used design patterns were selected for the study, each having peer-reviewed classification examples. These peer-reviewed classifications are given by nine projects that are part of the Pattern-like Micro-Architecture Repository (P-MARt[1] 04/10/19) [32]. Applications within the repository range from modeling and drawing tools to static analysis and refactoring frameworks, therefore its diversity is a good representation for the real world. Table I contains the distribution of design pattern instances across projects. *Original* are the mapping counts in the exploded form provided by the dataset. *Revised* are the mappings after a manual cleanup of the data. Some design pattern instances trace to classes not contained in the available source code repositories (e.g., java.awt, javax.swing, or third-party libraries) and were removed. Projects that did not include the target pattern were also not considered during the experiments (for the respective pattern) as controlling for imbalance (NPCR) is impossible, e.g., JRefactory for Template Method. At last, the Netbeans project was excluded as we were not able to parse it. Fixing the source code pieces in question was deemed futile as they were too numerous introducing potential bias in the evaluation. *Unique* are the unique mappings after the revision.

D. Procedures

First, 67 micro-structures were extracted (Section III-A) and possible candidates sampled (Section III-B) as shown in Figure 2. The micro-structure extraction and candidate sampling were based on our previous work [33]. Feature-Role

[1]http://www.ptidej.net/tools/designpatterns/index_html#2

Normalization (Section III-C) created the feature maps for each candidate and pattern. Cells within the FM were globally unique role identifiers ($id \in \mathbb{N}$) which received no further preprocessing such as rescaling or standardization. Further, no specific assignment strategy was employed for identifiers, i.e., role identifiers were assigned in their encountered order resulting in a total value range of $[0, 161]$ including the virtual roles.

The controlled variables from Section IV-A were executed in a systematic fashion resulting in the response variables from Section IV-B. All features (rows) were included. First, the FMs were collected into datasets, and negative-subsampling was applied according to the NPCR ratio (ExP 4). The dataset was then copied, permutated, and added k times (ExP 5). FMs were provided to CNNs as-is and linearized (row-wise) into a vector for RF. Then the models were fitted (ExP 3) and evaluated using *project-fold cross-validation* (ExP 7). This procedure was repeated 200 times in search of the optimal hyper-parameters (ExP 6) and for each pattern (ExP 1).

E. Experiment Results

The experiment parameters led to a total of 384 trials in which 76800 classifiers were fitted. The bulk of these classifiers were part of the optimization budget (ExP 6). In total, the best performing top-30 classifiers (including ties) for each Pattern, Model, and NPCR combination was selected for the evaluation. For instance, the boxplot in Figure 5 for the Singleton pattern at an NPCR of 1 represents the top-30 classifiers of this category. Furthermore, Table II contains the average test performance (marginalized over NPCR) for each pattern and model.

Using the Evans guidelines [34] for interpreting correlations, we see that CNN models have on average a *strong* performance with a Median (Med) 0.646 and an Interquartile Range (IQR) of 0.528 to 0.772. The worst median performance is given by the Template Method classifiers reaching a *moderate* $mcc_{Med} = 0.51;\ mcc_{IQR} = (0.43, 057))$. Best median performance is given by the Composite pattern with *strong* $mcc_{Med} = 0.79;\ mcc_{IQR} = (0.71, 0.85)$. The total variance in these estimates is acceptably low and would cause in the worst case a performance drop of $mcc_{\Delta} = 0.16$ degrading it to a *moderate* classifier. CNNs are rather robust against data imbalance with an average NPCR 1 to 10 change of $mcc_{\Delta} = -0.064$.

RF models are quite similar to CNN models for patterns with low role count but degrade quite drastically for patterns with four roles. RFs have on average a *moderate* performance with $mcc_{Med} = 0.48;\ mcc_{IQR} = (0.29, 0.64)$. At least a *moderate* ($mcc_{Med} = 0.47; mcc_{IQR} = (0.41, 0.52)$) up to a *strong* performance ($mcc_{Med} = 0.79; mcc_{IQR} = (0.67, 0.83)$) is given, excluding the Decorator models that were systematically worse ($mcc_{Med} = -0.35; mcc_{IQR} = (-0.38, -0.29)$) than a random model ($mcc = 0$). Again, the best performance as reached for Composite patterns with $mcc_{Med} = 0.79; mcc_{IQR} = (0.67, 0.83)$. On average, RF models exposed a variance of $mcc_{\Delta} = 0.14$ (Composite).

Figure 5 shows the effect of NPCRs on the classifier's performance. Both, CNN and RF suffer from imbalance except in the case of the Singleton pattern. CNNs are on average (all patterns) more robust against imbalance with an average NPCR 1 to NPCR 10 difference of $mcc_{\Delta} = 0.064$. However, they degrade for Composite and Decorator. The average difference for RFs is $mcc_{\Delta} = -0.16$.

Tests for independence [35] regarding MCC between CNN models with different permutation counts (ExP 5) were close to significant $\chi(3, 730) = 7.5245, p < 0.057$. Permutation passes for RF models were significant with $\chi(3, 745) = 75.117, p < 3.42^{-16}$. There was a significant effect concerning the NPCR for both model types with CNN ($\chi(5, N = 730) = 43.553, p < 2.85e^{-8}$) and RF ($\chi(5, N = 745) = 18.041, p < 0.002$).

V. Discussion

The presented results in Section IV-E fit with the intuition that 1) patterns with more roles are easier to detect, 2) models have a decline in performance with larger NPCR, 3) FMs fit well in the framework of CNNs.

Patterns with more roles inherently have more conditions that have to be met as each role describes some specific behavior a participating class must fulfill. This avoids *unintentional* implementations of these patterns and their detection because of a more elaborate structure and communication flow. Template Method, for example, describes a very general and freely used concept of abstract classes that defer implementation details of an algorithm to subclasses. Not only is this implementation technique used without actually having the intention to implement the Template Method pattern, but also pre-labeling of the dataset in this regard might prove challenging because of its general applicability. In contrast, Composite or Decorator use inheritance in conjunction with redirection techniques forcing the classifier to focus not only on inheritance but also the redirection aspect between multiple classes. Both patterns achieved excellent results that are quite robust across different NPCRs using CNNs (low degradation and low variance). However, RF, while similar to CNN for Singleton and Template Method, strongly degrades with patterns that have more roles (both variance and bias). This is to be expected as FMs encode relationship information in a 2-dimensional format (matrix) which needs to be serialized (loss of structural information) into a vector. Furthermore, serialization with patterns that have, e.g., four roles (Decorator), result in a $67 \times 4 = 268$ dimensional input vector. We did not explore the invertibility of the negative correlation (systematic mistakes) that RFs made for the Decorator. However, a simple inversion of the predictions might not generalize as expected and is open for further research. Nevertheless, RFs are still a good and fast solution for patterns with 1-3 three roles as the performance in Figure 5 shows.

The degrading performance of models with higher imbalance is often an issue. The minority class (class with lesser samples) is outweighed and too seldom encountered during training of the classifier in order to learn the essential aspects of it. Figure 5 shows that the performance degrades with higher NPCR. Still,

Table II
AVERAGE TEST-PERFORMANCE OF THE CONVOLUTIONAL NEURAL NETWORK (CNN) AND RANDOM FOREST (RF) BASED DETECTIONS.

Model	Pattern	Accuracy	Precision	Recall	F1	MCC
CNN	Singleton	0.88 ± 0.09	0.82 ± 0.12	0.73 ± 0.12	0.76 ± 0.10	0.72 ± 0.14
	Template Method	0.82 ± 0.09	0.63 ± 0.18	0.62 ± 0.20	0.58 ± 0.14	0.51 ± 0.12
	Composite	0.94 ± 0.02	0.86 ± 0.11	0.80 ± 0.14	0.82 ± 0.10	0.79 ± 0.10
	Decorator	0.81 ± 0.10	0.69 ± 0.15	0.80 ± 0.17	0.72 ± 0.13	0.60 ± 0.16
RF	Singleton	0.81 ± 0.07	0.63 ± 0.12	0.59 ± 0.12	0.59 ± 0.06	0.47 ± 0.06
	Template Method	0.85 ± 0.04	0.66 ± 0.21	0.58 ± 0.16	0.59 ± 0.15	0.49 ± 0.12
	Composite	0.93 ± 0.04	0.89 ± 0.10	0.71 ± 0.21	0.77 ± 0.16	0.79 ± 0.14
	Decorator	0.45 ± 0.06	0.10 ± 0.15	0.09 ± 0.13	0.09 ± 0.12	-0.35 ± 0.10

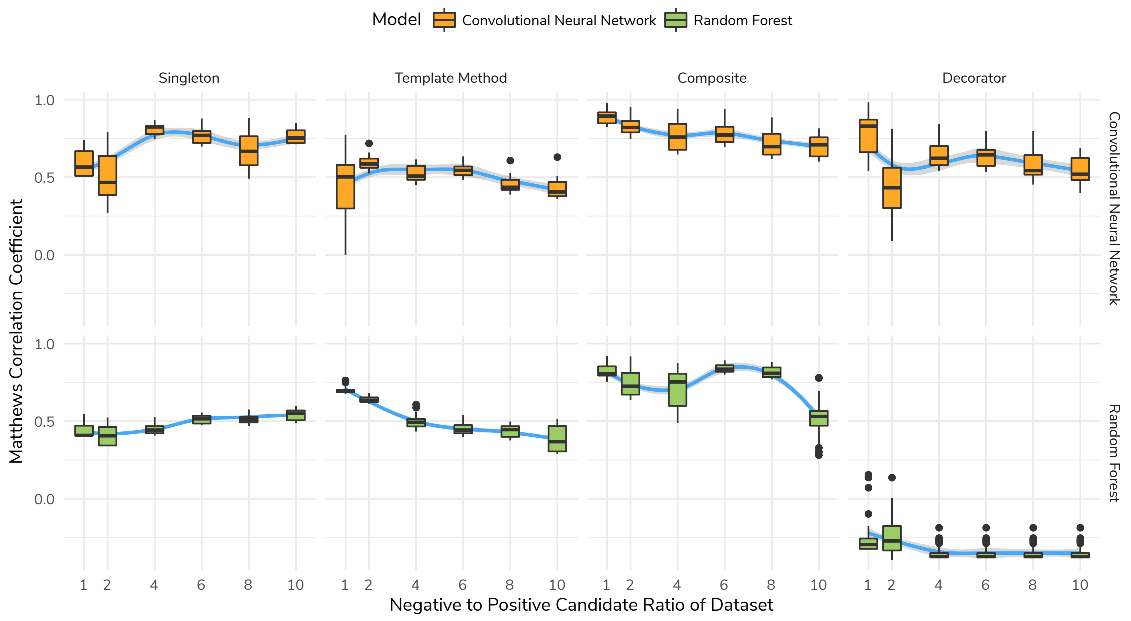

Figure 5. Each boxplot represents the top-30 fitted models for a given pattern and model, illustrating the trend across imbalanced datasets (NPCRs). Trends indicate that Feature Maps are a robust representation for DPD even in highly imbalanced settings.

the underlying trends are rather shallow except for Template Method. The increase in performance for the Singleton pattern is most likely related to the amount of available data. With only 13 examples it is expected that the classifiers use the extra negative samples to make the overall predictions more robust. In total, the degradation of performance over NPCRs is less extreme than initially anticipated. Though, it would need an additional study to attribute this effect properly.

FMs work well in the framework of CNNs because of their natural processing of volumes. CNNs are able to partially replicate the human DPD process via feature maps described in Section III-C. For example, a 3×3 filter can combine Aggregation, Inheritance, and Delegation across Component, Composite and Leaf in the first convolutional step from Figure 3. The result would be a higher level internal feature (of unknown quantity and semantics) representing this combination. Each layer then computes an even more abstract interpretation of the original feature leading to a good prediction. The same does not necessarily hold for RFs as the structural information (relationships between DP roles) would need to be recovered from the serialized vectors while still focusing on the content. We think that this natural advantage of CNNs resulted in the overall better performance especially for patterns with more roles.

Permutation passes increased the performance for RF classifiers and acted as additional regularization for CNNs without any performance loss (nor significant gain).

An advantage of NPCR (or generally imbalance) analysis is that it allows for direct comparison with other results via linear interpolation between NPCRs (as long as the datasets are the same). Zanoni et al. [15] conducted a similar study (same dataset) where they used multitudes of different classification models to find DPs. They report Accuracy for their *best* performing models without explicitly reporting the $NPCR$ values. We sketch a comparison by interpolating our results to match their $NPCR$ (computed form the ZeroR model) and compare our *average* model accuracies with their *best* model accuracies. Their best result for the *Singleton* pattern

($NPCR \approx 1.66$) was given by an RF model with $acc = 0.93$. Our RF is 20% worse with $acc = 0.73$, and the CNN is 19% worse with $acc = 0.77$. Their best *Composite* ($NPCR = 3$) model ($\nu - SVCRBF$ [36]) reaches $acc = 0.81$ where both of our models are better (CNN $acc = 0.93$ and RF $acc = 0.90$). Their best *Decorator* ($NPCR \approx 1.48$) model given by an RF with $acc = 0.82$ which is quite similar to our average CNNs with $acc = 0.79$. No data for *Template Method* is available. These results show how well FMs are suited for DPD given that we compared the average performance and not the best performing models. Besides, the authors used ordinary 10-fold cross-validation in which instances from the same unique mapping may leak information between evaluation and training folds.

VI. Threats to Validity

An internal threat to validity is given by the dataset (P-MARt 04/10/19 [32]) we used to train and evaluate our approach. We inspected many projects and their respective labeled design patterns in order to understand the misclassification during the experiments. Many of these projects are already outdated using old Java versions that forced the developers to take strange roundabouts in their implementations. Consequently, it is possible that some design patterns would be implemented differently today. For instance, the Singleton pattern is in modern systems most likely implemented via dependency injection frameworks. Furthermore, we could not understand some of the labeled pattern instances as they seem to diverge too far from the original design pattern definition.

The size of the dataset poses a threat, especially for the Singleton pattern as the model may overfit. To mitigate this thread we used repeated cross-validation to estimate the generalization performance on unseen data, added data augmentation (increases amount of observations), and included additional model specific regularization methods. Random forests naturally regularize with an increased number of trees. We used Dropout [37], kernel and activation regularization (L1, L2 [8], [7]), and early stopping [38] to regularize the CNNs.

Our experiment design tried to eliminate most external threats by using multiple patterns with different numbers of roles, with multiple bootstrapped datasets and different NPCRs. However, it only evaluates the performance of one pattern with k number of roles, and generalizing it to all patterns with k roles may only be possible on a limited scale. Nevertheless, the broad sample of models still provides valuable insight into the methodologies performance in many different settings.

VII. Related Work

The DPD research community has a long and active history with various approaches, tools, and methodologies. Several interests can be distilled from DPD and that are reflected in Figure 2. A coarse classification of these interests is feature extraction, intermediate representations, candidate sampling, and inference methods. This work focused on intermediate representations and inference methods.

Zanoni [15], with whom we compared our results, uses micro-structures, clustering, and various machine learning approaches to find design patterns. Along with Arcelli, they proposed a framework that uses these techniques called MARPLE [18]. Most influential work for feature maps and our approach is given by Tsantalis et al. [17], in which they extract adjacency matrices from the ASG reflecting a specific aspect. An example of an aspect would be Generalization or Association, and in such a way they may be seen as micro-structures. The approach itself compared the adjacency matrices from patterns with the matrices extracted from the system via an iterative similarity scoring algorithm proposed by Blondel et al. [39]. Their approach provides nearly always a recall and precision of 1 for the patterns detected in this work. However, their evaluation is only based on three manually inspected projects (JHotDraw, JRefactory, and JUnit). The authors accounted for subjective bias through manual inspection but did not mitigate its possible impact. Finally, they report that their methodology suffers from computational inefficiency caused by the (adjacency) matrices and the matching algorithm.

One big issue in the DPD community is that results are hard to reproduce and compare because of the multistage nature of detectors. Fontana et al. [40] give an attempt to make results comparable. Nevertheless, the approach is a community-driven web-based benchmarking system that suffers, like many other community-based approaches, from the cold start phenomenon, i.e., it provides not enough upfront benefit to justify its usage in the first place. To improve this situation, we employed an evaluation strategy that decouples the last stage of the DPD pipeline simplifying the reproducibility and comparability of the results to the availability of the dataset that candidate samplers produce. Furthermore, we provided a comprehensive approach to evaluate detectors in a more general fashion than peak performance models.

VIII. Conclusion

This work presented Feature Maps (FMs), how they are computed via Feature-Role Normalization (FRN) and used for design pattern detection. Feature maps themselves can be understood as flexible and comprehensible source code representation useful beyond DPD. For DPD, FMs provide a representation that allows for robust detection performance even if the datasets are strongly imbalanced. In conclusion, FMs do not only help to extract the information developers weave into their systems, but also provide means to represent and comprehend them in a compact fashion.

In the future, we are planning to extend the study on FMs in the context of DPD by applying the methodology to more design patterns and a bigger dataset. Futhermore, we are planning to compare the methodology to learning algorithms that handle graph representations natively.

Acknowledgments

The research reported in this paper has been supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

References

[1] E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software / Erich Gamma ... [et Al.]*, ser. Addison-Wesley professional computing series. Addison-Wesley.

[2] L. Breiman, J. Friedman, R. Olshen, and C. J. Stone, "Classification and regression trees."

[3] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," vol. 86, no. 11, pp. 2278–2324.

[4] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization." [Online]. Available: http://arxiv.org/abs/1412.6980

[5] C. M. Bishop, *Pattern Recognition and Machine Learning*, ser. Information science and statistics. Springer.

[6] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. Springer New York.

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, http://www.deeplearningbook.org.

[8] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*, ser. Adaptive computation and machine learning series. MIT Press.

[9] S. Geisser, *Predictive Inference: An Introduction*. Chapman & Hall.

[10] B. W. Matthews, "Comparison of the predicted and observed secondary structure of T4 phage lysozyme," vol. 405, no. 2, pp. 442–451.

[11] D. M. W. Powers, "Evaluation: From precision, recall and F-factor to ROC, informedness, markedness and correlation."

[12] S. Alhusain, S. Coupland, R. John, and M. Kavanagh, "Towards machine learning based design pattern recognition," pp. 244–251.

[13] A. Chihada, S. Jalili, S. M. H. Hasheminejad, and M. H. Zangooei, "Source code and design conformance, design pattern detection from source code by classification approach," vol. 26, pp. 357–367.

[14] S. Hussain, J. Keung, and A. A. Khan, "Software design patterns classification and selection using text categorization approach," vol. 58, pp. 225–244.

[15] M. Zanoni, F. Arcelli Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," vol. 103, pp. 102–117.

[16] A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery through visual language parsing and source code analysis," vol. 82, no. 7, pp. 1177–1193.

[17] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design Pattern Detection Using Similarity Scoring: Software Engineering, IEEE Transactions on," vol. 32, no. 11, pp. 896–909.

[18] F. Arcelli Fontana and M. Zanoni, "A tool for design pattern detection and software architecture reconstruction," vol. 181, no. 7, pp. 1306–1324.

[19] H. Thaller, "Towards Deep Learning Driven Design Pattern Detection." [Online]. Available: http://resolver.obvsg.at/urn:nbn:at:at-ubl:1-11857

[20] S. Maggioni, Stefano Maggioni, and S. Maggioni, "Design Pattern Detection and Software Architecture Reconstruction: An Integrated Approach based on Software Micro-structures."

[21] J. M. C. Smith, *Elemental Design Patterns*. Addison-Wesley, vol. 17, no. 2. [Online]. Available: http://evi.sagepub.com/cgi/doi/10.1177/1356389011400889

[22] F. A. Arcelli Fontana, M. Zanoni, and S. Maggioni, "Using design pattern clues to improve the precision of design pattern detection tools," vol. 10, no. 1, pp. 1–31.

[23] J. Y. Gil and I. Maman, "Micro Patterns in Java Code," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. ACM, pp. 97–116.

[24] F. Arcelli Fontana, S. Maggioni, and C. Raibulet, "Understanding the relevance of micro-structures for design patterns detection," vol. 84, no. 12, pp. 2334–2347.

[25] R. J. Wilson, *Introduction to Graph Theory*. John Wiley & Sons, Inc.

[26] L. Babai and E. M. Luks, "Canonical labeling of graphs." ACM Press, pp. 171–183.

[27] C. J. C. Burges, "A Tutorial on Support Vector Machines for Pattern Recognition," vol. 2, no. 2, pp. 121–167.

[28] M. Gori, M. Maggini, and L. Sarti, "A recursive neural network model for processing directed acyclic graphs with labeled edges," vol. 2, pp. 1351–1355.

[29] S. Hochreiter and J. Urgen Schmidhuber, "Long Short-Term Memory," vol. 9, no. 8, pp. 1735–1780.

[30] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," pp. 1–14.

[31] "Design pattern detection using software metrics and machine learning."

[32] Y.-g. Guéhéneuc, "PMARt: Pattern-like Micro Architecture Repository." [Online]. Available: http://www-etud.iro.umontreal.ca/ptidej/Publications/Documents/EuroPLoP07PRa.doc.pdf

[33] H. Thaller, *Hannes Thaller*. [Online]. Available: hannes-thaller.comhttp://www.hannes-thaller.com/

[34] J. D. Evans, *Straightforward Statistics for the Behavioral Sciences*. Brooks/Cole Pub. Co.

[35] W. H. Kruskal and W. A. Wallis, "Use of Ranks in One-Criterion Variance Analysis," vol. 47, no. 260, pp. 583–621.

[36] A. J. Smola and B. Schölkopf, *A Tutorial on Support Vector Regression*.

[37] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," vol. 15, no. 1, pp. 1929–1958. [Online]. Available: http://dl.acm.org/citation.cfm?id=2627435.2670313

[38] L. Prechelt, "Early Stopping-But When?" in *Neural Networks: Tricks of the Trade, This Book Is an Outgrowth of a 1996 NIPS Workshop*. Springer-Verlag, pp. 55–69. [Online]. Available: http://dl.acm.org/citation.cfm?id=645754.668392

[39] V. D. Blondel, A. Gajardo, M. Heymans, P. Senellart, and P. van Dooren, "A Measure of Similarity Between Graph Vertices: Applications to Synonym Extraction and Web Searching," vol. 46, no. 4, pp. 647–666.

[40] F. Arcelli Fontana, A. Caracciolo, and M. Zanoni, "DPB: A Benchmark for Design Pattern Detection Tools," in *2012 16th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 235–244.