A Hybrid Graph Analysis and Machine Learning Approach Towards Automatic Software Design Pattern Recognition Across Multiple Programming Languages

Roy Oberhauser
Computer Science Dept.
Aalen University
Aalen, Germany
e-mail: roy.oberhauser@hs-aalen.de

***Abstract*—The volume of program source code created, reused, and maintained worldwide is rapidly increasing, yet code comprehension remains a limiting productivity factor. For developers and maintainers, well known common software design patterns and the abstractions they offer can help support program comprehension. However, manual pattern documentation techniques in code and code-related assets such as comments, documents, or models are not necessarily consistent or dependable and are cost-prohibitive. To address this situation, we propose the Hybrid Design Pattern Detection (HyDPD), a generalized approach for detecting patterns that is programming-language-agnostic and combines graph analysis (GA) and Machine Learning (ML) to automate the detection of design patterns via source code analysis. Our realization demonstrates its feasibility. An evaluation compared each technique and their combination for three common patterns across a set of 75 single pattern Java and C# public sample pattern projects. The GA component was also used to detect the 23 Gang of Four design patterns across 258 sample C# and Java projects as well as in a large Java project. Performance and scalability were measured. The results show the advantages and potential of a hybrid approach for combining GA with artificial neural networks (ANN) for automated design pattern detection, providing compensating advantages such as reduced false negatives and improved F_1 scores.**

Keywords—software design pattern detection; machine learning; artificial neural networks; graph analysis; software engineering.

I. INTRODUCTION

This paper extends our previous work on automatic design pattern detection (DPD) [1].

A major digitalization transformation is underway throughout industry and society [2], dependent on increasing amounts of software to drive it. For instance, Google is said to have at least 2bn lines of code (LOC) accessed by over 25K developers [3], and GitHub currently reports over 200m repositories and 73m developers [4]. It has been estimated that worldwide well over a trillion LOC exist [5] with 111b lines of new software code created annually [6]. This is exacerbated by a limited supply of programmers and high employee turnover rates for software companies, e.g., 1.1 years at Google [7]. Furthermore, the high degree of utilization in live business operations creates additional time pressure and stress for rapid turnaround, development or maintenance cycles, or deployment times.

The economics of rapidly growing codebases, code longevity, and high turnover make program development and maintenance challenging programmers (herewith including maintainers) especially with regard to fast program comprehension and understanding of (legacy) codebases. Given limited resources and such a vast amount of code, ~75% of technical software workers are estimated to be doing maintenance [8]. Moreover, program comprehension may consume up to 70% of the software engineering effort [9]. Activities involving program comprehension include investigating functionality, internal structures, dependencies, run-time interactions, execution patterns, and program utilization; adding or modifying functionality; assessing the design quality; and domain understanding of the system [10]. Code that is not properly understood by programmers impacts efficiency and reduces quality.

For program comprehension, experts tend to develop efficiently organized specialized schemas or abstractions that contribute to efficient problem and system decomposition and comprehension, and macrostructures (or chunks) and beacons (or cues) being important elements in cognition mental models [11]. In the area of software engineering, software design patterns have been well-documented and popularized, including the Gang of Four (GoF) [12] and POSA [13]. The application of abstracted and documented solutions to recurring software design problems has been a boon to improving software design quality and efficiency. Discovering such common macrostructures or associated pattern terminology in code can serve as beacons to such abstracted macrostructures and may help identify aspects such as the author's intention or purpose.

However, the actual detection and post-coding documentation of these software design patterns remain a challenge. As design patterns have mostly been described informally, their implementation can vary widely, depending on various factors such as the programming language, the natural language and keywords used, the concrete pattern structure, the terminology awareness of the programmer, their experience, and their understanding and (mis)interpretation. Furthermore, the pattern books referenced above were published over 25 years ago and not standardized with regular updates. Pattern variants may occur and patterns may evolve over time with technology. Additionally, the manual documentation of software design patterns usage in project documents such as the architecture specification may not be dependable due to inconsistencies with the codebase, e.g., prescriptive documentation of intentions, adaptations during development, or maintenance changes. Determining actual pattern usage can be beneficial for identifying which patterns

are used where and can help avoid unintended pattern degradation and associated technical debt and quality issues. However, the investment necessary for manual pattern extraction, recovery, and archeology is cost prohibitive and not sustainable due to the high design competency and labor-intensive code analysis effort required, especially in light of the aforementioned codebase sizes and high turnover. Prechelt et al. [14] come to the conclusion that explicit identification of patterns in code (here via manually place pattern comment lines) facilitate faster and less error-prone maintenance tasks.

In source code, patterns may not be explicitly mentioned or commented at all, or they might be or inconsistently or incorrectly mentioned. Semantic issues due to various natural languages and naming differences may also cause beacons or keywords to differ. Since in our context we assume access to source code, intentional obfuscation via a tool is unlikely, but unintentional obfuscation is possible. One way to nevertheless explicitly identify patterns in code would be automated detection via a tool. Yet automated detection and extraction of software design patterns from code is not readily available among popular software development tools. Various research work has attempted to find automated techniques, yet these often fail to recognize or address coverage of all of the basic 23 GoF patterns and rather emphasizing certain design patterns, or were not evaluated on a larger code base. They conclude by suggesting that a combination of techniques might be promising research approach.

In our previous work DPDML (Design Pattern Detection using Machine Learning) [1], we showed the feasibility of our cross-language approach for DPD by realizing the ML core of our approach. Our evaluation using 75 unique Java and C# code projects for training and testing to detect three different types of GoF patterns (creational, structural, and behavioral) provided insights into its potential and limitations. It necessitated finding sufficient training sets (sample projects) for each pattern, and our realization, while combining metrics and semantic analysis, relied on a single technique, namely ML.

This paper contributes our hybrid automated design pattern detection approach called Hybrid Design Pattern Detection (HyDPD) supporting multiple programming languages and amalgamating GA with ML to utilize the advantages of both techniques while decreasing their liabilities. Our realization of the solution approach shows its feasibility. An evaluation compared each technique and their combination for three common patterns across a set of 75 single-pattern Java and C# public sample pattern projects. Furthermore, to provide insights into its potential and limitations, the GA component was applied on 23 GoF design patterns across 258 sample C# and Java projects, as well as a larger Java project (JUnit).

The structure of this paper is as follows: the following section discusses related work. Section 3 describes our solution approach. In Section 4, our realization is presented, which is followed by our evaluation in Section 5. Thereafter, a conclusion is provided.

II. Related Work

Various approaches have been used for software design pattern detection, and they can be categorized based on different analysis styles, such as structural, behavioral, or semantic (and some utilizing a combination of styles) [15]. Structural analysis utilizes static DPD based on inter-class dependencies, data types, and method invocations found in code. Behavioral analysis extracts behavior via static and/or dynamic analysis techniques, since structure alone may not suffice to differentiate patterns. Semantic analysis utilizes naming and annotations to distinguish patterns. Another categorization option is to group by detection methodologies or techniques (learning-based, graph-based, metric-based, etc). As a consequence, some work may fall into multiple categories.

Graph-based approaches include: Yu et al. [16] that reverse engineer code to UML class diagrams and from XMI parse and analyze sub-patterns with class-relationship directed graphs. Mayvan and Rasoolzadegan [17] use a UML semantic graph. Bernardi et al. [18] apply a DSL-driven graph matching approach. DesPaD [19] extract the abstract syntax tree from the code, create a single large graph model of a project, and then apply an isomorphic sub-graph search method using the Subdue tool. Further isomorphic subgraph approaches include Pande et al. [20] and Pradhan et al. [21], both of which begin with UML class diagrams.

Learning-based approaches map the DPD problem to a learning problem, and can involve classification, decision trees, feature maps or vectors, Artificial Neural Networks (ANNs), Convolutional Neural Networks (CNNs), Support Vector Machines (SVMs), etc. Examples include Alhusain et al. [22], Zanoni et al. [23], Galli et al. [24], Ferenc et al. [25], Uchiyama et al. [26][27], and Dwivedi et al. [28]. Thaller et al. [29] describe a micro-structure-based structural analysis approach based on feature maps. Chihada et al. [30] convert code to class diagrams, which are then transformed to graphs, and have experts create feature vectors for each role based on object-oriented metrics and then apply ML.

Additional approaches include: reasoning-based approaches such as Wang et al. [31] that is based on matrices. Examples of fuzzy logic approaches include Alhusain et al. [32] and Hussain et al. [33]. Examples of rule-based approaches include Sempatrec [34] and FiG [35], which uses an ontology representation. Metric-based approaches include MAPeD [36], PTIDEJ [37], Uchiyama et al. [26][27], and Dwivedi et al. [28]. Fontana et al. [38] analyze microstructures based on an abstract syntax tree. An example semantic-analysis style approach is Issaoui et al. [39]. DP-Miner [40] uses a matrix-based approach based on UML for structural, behavioral, and semantic analysis.

The DPD styles and methodologies used are quite fractured and none has reached a mature and high-quality result with an accessible and executable implementation that we could evaluate. We are not aware of any approach yet that can automatically and reliably detect all 23 GoF design patterns. Most have some limitation or drawback, and the success rate reported among the approaches varies tremendously. Thus, further investigation and research in this

area is essential to enhancing the knowledge surrounding this area. In contrast to the aforementioned work, our HyDPD solution offers a hybrid code-centric approach combining various promising structural, behavioral, and semantic analysis techniques to leverage the strengths of each. In contrast to others, it supports multiple popular programming languages. With HyDPD we show the advantages of combining GA, ML, metrics, and semantic analysis.

III. Solution

Of all available artifacts for DPD, source code represents the reality rather than the intention, and should be readily available, whereas other pattern information (binaries may not build, and runtime instrumentation, UML models, or documentation may not exist). Furthermore, as UML diagrams can be generated by tools from the source code, the underlying data they utilize is also available in the source code. As shown in Figure 1, our solution approach thus begins with source code as the input and is based on the following principles:

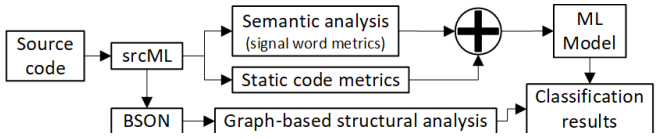

Figure 1. The HyDPD solution concept.

Programming language-independent: since patterns are independent of programming language, our solution abstracts the source code by converting it into an abstracted common format for further processing. For this, our realization currently utilizes srcML [41], which provides an XML-based format, currently supporting C, C++, Java, and C#. If other abstract syntax formats are standardized and available for analysis in a common format, these also can be considered.

Semantic analysis: common pattern signal words in the source code can be used as an indicator or hint for specific pattern usage. Additional natural languages can be supported to detect usage of pattern names or their constituent components in case they were coded in other languages. Our realization supports German, Russian, and French.

Static code metrics extraction: various static code metrics are utilized to detect and differentiate design patterns. The value ranges of metrics are normalized to a scale of 0-1 for utilization with an ANN.

ML model: in utilizing ML to analyze sample data, a model can learn how to classify new unknown data, in our case to differentiate design patterns. Our realization may apply or combine any ML model that suits the situation. Currently, an ANN is used because we were interested in investigating its performance, and intend in future work to detect a wide pattern scope, pattern variants, and new patterns. From our standpoint, alternative non-ML methods such as creating a rule-based system by hand would require labor and expertise as the number of patterns increases and new undiscovered patterns should be detected. With an appropriate ML model, these should be learned automatically and be more readily detected. Challenges include appropriate slicing of the codebase for appropriate metrics and pattern comparison, and finding suitable and large enough training datasets for each pattern.

Graph-based structural analysis: the XML-based code representation is converted to a BSON (Binary JSON) format and stored in a graph database to support graph-based structural analysis. In contrast to ML, the advantages include the ability to apply graph queries across an entire codebase and not requiring any training data. Liabilities include the need for hand-crafted detection queries that are not too specific (thus overlooking many with only slight variations) nor too general or ambiguous to be of practical use (identifying too many false positives).

The underlying hypothesis driving our HyDPD investigation is that amalgamating additional data and metrics in combination with various analysis techniques such as graph and ML models results in better classification accuracy compared to any single technique or metric alone. From a practicality standpoint, our approach could reduce the labor involved in detecting and documenting patterns compared to finding potential patterns manually by perusing code and accurately classifying them (e.g., for assessing or modernizing an unfamiliar legacy system), and can assist developers, maintainers, or experts involved in various software archeology activities.

IV. Realization

The realization of the HyDPD approach consists of two main components: HyDPD-ML that applies the ML technique and HyDPD-GA that applies the GA technique. Python was used to implement the prototype due to its versatility and large selection of available libraries, while a Jupyter Notebook was used for the DPD user interface.

A. HyDPD-ML

Python was used to implement the prototype due to its versatility and the available libraries to support the implementation of ANNs. TensorFlow was chosen along with Keras as a top-layer API.

For ML, a sufficient dataset of different and realistic projects was needed to support supervised learning. While certain pattern examples in code can readily be found, finding a larger set of different ones in different project settings and programming languages turns out to encounter various practical challenges and is labor intensive. Due to resource and time constraints, our ML realization thus initially focused on having the network learn to detect one pattern out of each of three main pattern categories: from the structural category - Adapter; from the creational patterns -Factory; and from the behavioral patterns - Observer. Future work will expand the pattern scope.

Metric-based matching: The ElementTree parser was used to traverse srcML and count the specific XML-tags. The metric values were not separated by roles or classes, but are merged and evaluated as a whole. The metrics used were inspired by Uchiyama et al. [26] and are shown in Table I.

TABLE I. OVERVIEW OF METRICS

Abbreviation	Description
NOC	Number of classes
NOF	Number of fields
NOSF	Number of static fields
NOM	Number of methods
NOSM	Number of static methods
NOI	Number of interfaces
NOAI	Number of abstract interfaces

TABLE II. SIGNAL WORDS FOR DESIGN PATTERNS

Pattern	Signal Words			
Adapter	Adapter	adaptee	target	adapt
Factory	Factory	create	implements	type
Observer	observer	state	update	notify

Semantic-based matching: An obvious approach to pattern detection is naming. If a developer already used common design pattern terminology in the code, then this should be utilized as a pattern detection indicator. For our signal word detection, we translated the signal words to German, French, and Russian to improve results for non-English code.

Semantic variations: To determine if other signal words beyond the design pattern name were used in implementations, we analyzed several examples of implemented design patterns. 12 additional signal words were selected, four for each pattern as shown in Table II.

Internationalization: To test internationalization, the Python library *translate* was used to translate the signal words to German, French, and Russian. Rather than extending the list of metrics passed to the ANN, a match with a translated word is counted in the same input parameter as the original English words. Applying Natural Language Processing (NLP) to reduce words by stemming or creating lemmas to compare to a defined word list would also be possible, and may improve or deteriorate the results, if for instance the input array contained further zeros when no signal words were found.

ANN: Based on our realization scope, since the input array is not multidimensional, deep neural networks (DNNs) with additional layers would not necessarily yield improved results. We thus chose to realize an Artificial Neural Network (ANN) with one input layer, two hidden layers, and one output layer as shown in Figure 2. We created the network with the Keras API with the TensorFlow Python library.

The input layer size matches the data points, and as there are 7 metrics and 12 semantic match values, this makes 19 input values total. The input model structure is a numpy array as follows:

```
[NOC, NOF, NOSF, NOM, NOSM, NOI, NOAI, ASW1,
  ASW2, ASW3, ASW4, FSW1, FSW2, FSW3, FSW4,
          OSW1, OSW2, OSW3, OSW4]
```

The first 7 values correspond to Table I while the rest indicate the number of signal word matches from Table II. SW=Signal Word, A=Adapter, F=Façade, and O=Observer, 1-4 implies the corresponding table column. Only 7 metric values are utilized when no signal words exist.

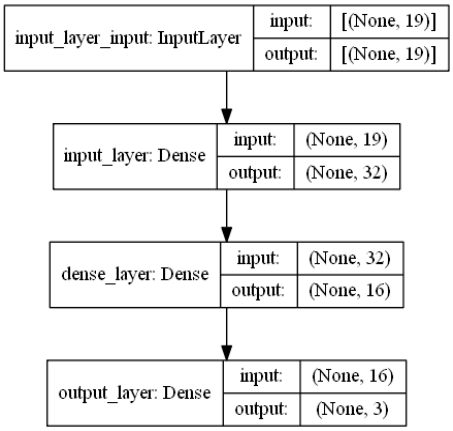

Figure 2. ANN model overview created with Keras.

The first hidden layer is a dense layer (with each neuron fully connected to the neurons in the prior layer) consisting of 32 neurons. The activation function was a rectified linear unit (ReLU). The second layer is a dense layer with 16 neurons. This conforms with the general guideline to gradually decrease the neurons as one approaches the output layer. The output layer consists of three neurons to match the three design patterns that should be detected. The "Softmax" activation method is used, which is often used in classification problems and supports identifying the confidence of the network in its decision. The "Adam" algorithm is a universal optimizer that is recommended in a wide assortment of papers and guides. As no specialized optimizer was needed, "Adam" with its default values was chosen as defined in [42]. No regularization was applied in each layer. Adam automatically adjusts and optimizes the learning rate. Sparse categorical cross entropy was applied as the loss function for this multi-class classification task.

The size of the ANN should fit the size of the problem. Small adjustments to the ANN structure showed no significant performance impact, whereas significantly increasing the neuron count or layer count negatively impacted results. With two hidden layers and 48 neurons, the first layer contains 640 parameters, the second layer 528, and the output layer 51, resulting in 1219 parameters that are adjusted during training.

The network is trained in epochs, wherein the complete training set is sent through the network with weights adjusted. As the weights and metrics change per epoch, an early-stopping callback stops the training if the accuracy of the network decreases over more than 10 epochs, saving the network that had the best accuracy. A validation dataset is typically used during training to monitor results on unlearned data after each epoch, but as our training set was limited, we used a prepared testing dataset with known labels.

1) Training Datasets

As to possible design pattern training sets, the Pattern-like Micro-Architecture Repository (P-MARt) includes a collection of microstructures found in different repositories such as Jhotdraw and JUnit. However, because these patterns are intertwined with each other, they do not provide isolated example specimens for training the ANN. The Perceptrons

Reuse Repositories could theoretically provide many instances of design patterns for a training dataset, but no results were provided on the website during the timeframe of our realization, and while the source code analyzer is free, the servers could not be reached.

We did manage to find training data as detailed in the next section. Since our initial intent for HyDPD was a much broader scope for data pattern mining, and because we expected a large supply of sample data, we focused on an ANN realization. We were also interested in determining if we could train an ANN to detect these patterns with relatively few samples. However, due to unexpected additional resource and time constraints involved in finding pattern samples manually, we had to reduce the number of design patterns involved, and could not compare the ANN with alternative classification schemes such as Naïve Bayes, Decision Tree, Logistic Regression, and SVMs, but intend to in future work.

B. HyDPD-GA

For the GA realization, the srcML is converted to BSON and stored in MongoDB. A Neo4j Cypher procedure is then used to import the BSON from MongoDB into the Neo4j database. `OPTIONAL MATCH` is used to permit variations of patterns to be in the result set, while missing entities are notated with `None`. The result set is provided as a `Dataframe` structure. Figure 3 shows the Python classes used for the implementation. Not all methods in the Python project are depicted in the diagram. Some methods have been defined in the scope of view of the class methods. They are only used within the framework of the respective method and serve to improve the readability of the code. For example, the method `resolve_names` in the `ResolveJsonTypes` class contains another 24 methods.

The `FileStorage` class is required to create the folders in advance in which the intermediate results of program execution are stored. All classes other than `DPDetector` use the `Settings` class to query the values of the current configuration parameters (e.B. Uniform Resource Locator (URL) of the MongoDB). The `SrcmlDriver` class is used to convert the code project into the srcML XML representation. The `SrcmlJsonConverterMulti` class is used to detect whether the specified code project was written in Java or C#. Depending on this, either the class `SrcmlJsonConverterForCSharp` or `SrcmlJsonConverterForJava` is used to convert the XML representation of the project to JSON. Both classes inherit from the abstract class `SrcmlJsonConverter`, which contains shared functionality. In addition, the `SrcmlJsonConverter` class interacts with the `ResolveJsonTypes`, which undertakes part of the conversion process. In addition, all classes (except `ResolveJsonTypes`) use the `SrcmlUtil` class, which supports parsing the XML representation of the project. The `TypesNeo` class interacts with the `TypesMongo` class to import a specific project from MongoDB into the Neo4j database. The `DPDetector` class uses the `TypesNeo` class when executing the DPD queries on the Neo4j database.

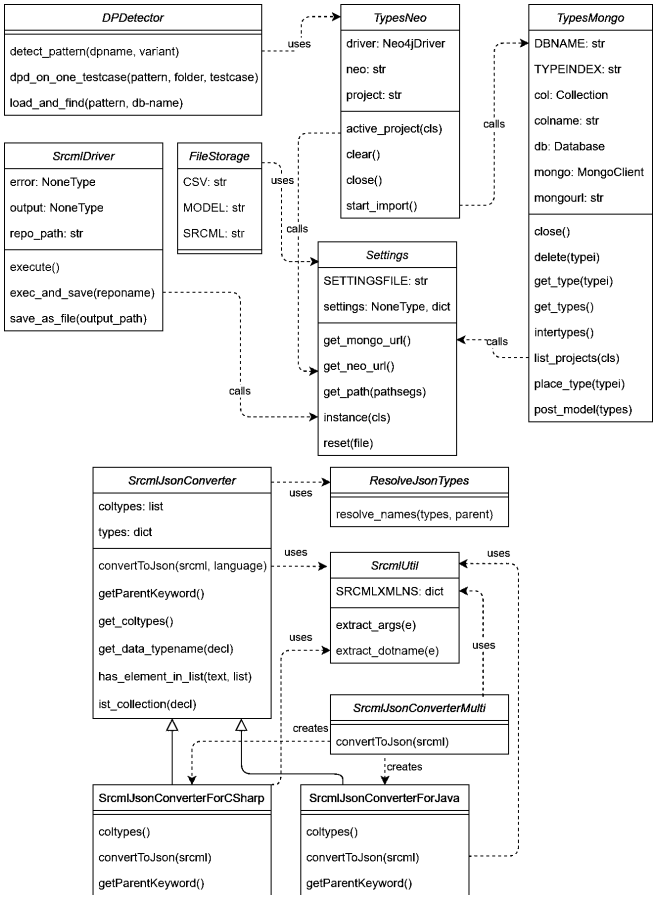

Figure 3. HyDPD-GA Python classes (partial view).

For each parsed class, the full class name, available imported entities, and inheritance from other classes and interfaces are extracted. A flag indicates whether the respective class is abstract. For each attribute, the type, name, and two flags are used for indicating a static attribute or if it involves a collection. The parsed class contains a collection of methods with their implementation, abstract methods, constructors, and getter/setter methods taken into account. For each method, the following data is extracted:

- method name;
- static flag;
- abstract flag;
- constructor flag;
- return type;
- flag whether the method is accessible outside its class;
- method arguments;
- declared variables including name, type, and collection flag;
- type assigned when a variable was initialized; and
- names of other methods called from this method.

After all relevant information has been extracted and stored in the form of the JSON object in the `types` attribute, the second stage of processing takes place. This consists in resolving the name via all possible types and methods. For this, the static class `ResolveJsonTypes` or its method `resolve_names` is utilized. All possible types refers to the

inherited classes, class attributes, method variables, method arguments, and the type a method returns. Resolving methods considers the methods that are called as well as the methods that are used for initialization, and is based on the imported entities, the attributes and methods of existing and inherited classes, and the local variables. Moreover, the relationship of overriding a parent method via a method in the class being analyzed is included. In addition, the attributes and methods of the parent class are added to the inheriting class.

The method is then executed `transform_types` to convert the obtained JSON format to the mongoDB BSON format. The result is returned by the `convertToJson` method in the `SrcmlJsonConterverMulti` class. To store the BSON object in MongoDB, a new object of the `TypesMongo` class is instantiated. It then calls the method `transformed_types` and passes the BSON object as input to that method. Finally, the code project is saved in BSON format in MongoDB.

The class `DPDetector` performs pattern recognition. The method `dpd_on_one_testcase` in this class removes any preexisting Neo4j entities and searches for a single pattern in the specified project. It instantiates a new object of the class `TypesNeo`, importing the corresponding collection from MongoDB into the Neo4j database using a Cypher query. The `detect_pattern` method uses a Cypher query to search for a designated design pattern. If matches with the query exist, the result records are grouped by the main participant of the pattern, and the correctness of the match is calculated as a proportion of the matching nodes relative to the total number of nodes searched for (e.g., 0.70 would indicate a 70% query match). The resulting records are then returned as a `DataFrame`.

An example Cypher query for the Chain of Responsibility (CoR) pattern is shown in Figure 4. First, the type `handler` is defined that has two methods: `handle_op` and `set_op`. The `set_op` has at least one argument of type `handler`. Furthermore, two types `c_handler` and `c_handler2` that inherit from the `handler` are defined. In addition, both classes must have a `handler` attribute and a method that overrides the parent class's method.

There should also be a `client` type with a method `client_op`. The `client_op` method should call either `handle_op` or `c_handle_op`. In the following, negative conditions are defined that further refine the query and distinguish it from other patterns. The `client` type cannot inherit from the `handler` type. The type `clientc_handler` and `c_handler2` must not have a collection of the type `handler`.

The Cypher query for each pattern was tuned as follows: For each pattern, a test dataset of at least six Java and C# examples from an internet search of GitHub and other sources was used to tune the Cypher query in such a way that it detects the expected pattern (true positives or TPs). In case it did not, the code was analyzed to determine the underlying cause and, if possible, the query or, if appropriate due to a faulty implementation of the pattern, the code adjusted until it is detected. If not, the underlying cause was identified; reasons included for instance 1) a non-compliant application of the pattern that violate core structural rules of the pattern (intentionally or unintentionally due to an author's misunderstanding), 2) referencing external classes (missing) that were not contained in the source code and required to fulfill the pattern, 3) the use of functional programming constructs.

```
MATCH
(handler:Type:Abstract)-[:HAS]->(handle_op:
  Operation),
(handler)-[:HAS]->(set_op:Operation),
(set_op)-[:HAS_ARG]->(handler),

(c_handler:Type)-[:INHERITS*1..3]->(handler),
(c_handler)-[:HAS]->(handler),
(c_handler)-[:HAS]->(c_handle_op:Operation),
(c_handle_op)-[:OVERRIDES*1..3]->(handle_op),

(c_handler2:Type)-[:INHERITS*1..3]->(handler),
(c_handler2)-[:HAS]->(handler),
(c_handler2)-[:HAS]->(c_handle_op2:Operation),
(c_handle_op2)-[:OVERRIDES*1..3]->(handle_op),

(client:Type)-[:HAS]->(client_op:Operation)

WHERE ((client_op)-[:CALLS]->(handle_op)
OR (client_op)-[:CALLS]->(c_handle_op))
AND NOT (client)-[:INHERITS]->(handler)
AND NOT (c_handler)-[:HAS {collection: true}]->(
  handler)
AND NOT (c_handler2)-[:HAS {collection: true}]->(
  handler)

RETURN *
```

Figure 4. Cypher query for the Chain of Responsibility pattern.

Thereafter, each query was executed on all 22 other patterns to determine what should not belong to the query result with regard to FPs. If a positive in an unexpected pattern was detected, it was analyzed to determine if 1) the query must be further tuned, 2) something in the target code example is inappropriate (abstract method, a participant is not allowed in an inheritance hierarchy, optional pattern elements), etc.

An example excerpt from the result output of a Jupyter Notebook is shown in Figure 5.

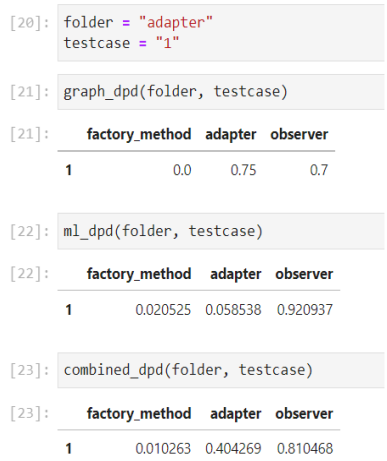

Figure 5. Example Jupyter Notebook result output excerpt.

C. HyDPD

To enhance DPD, the HyDPD realization combines both HyDPD-ML and HyDPD-GA components, with each component supplying a probability *P* as shown in (1) about the likelihood of a certain pattern being detected, where *w* is the weighting. *w* is currently equal and set to 0.5 until further empirical insights are gathered as to which approach is typically more accurate.

$$P = \omega_{ML} * P_{ML} + \omega_{GA} * P_{GA} \quad (1)$$

Weight Tailoring: The weightings can be tailored across all patterns or on a per pattern basis, for instance based on empirical accuracy rates if one technique is determined to have better accuracy for a specific design pattern.

V. EVALUATION

Since the primary contribution in this paper is the new GA component and its hybrid combination with ML, this evaluation focused primarily on investigating the potential of GA and its utilization in combination with ML. Our research questions (**RQ**s) are adjusted to the limitations of our available datasets and time and resources.

The first three RQs utilize three common GoF patterns for analyzing and comparing the HyDPD components and the hybrid approach, since HyDPD-ML requires larger pattern-specific datasets for training:

RQ1. *How does HyDPD-ML perform against three common GoF patterns?*

RQ2. *How does HyDPD-GA perform against three common GoF patterns, and how does it compare with HyDPD-ML?*

RQ3. *How does the hybrid HyDPD perform against three common GoF patterns, and how does it compare with HyDPD-ML and HyDPD-GA?*

The next four RQs focus on analyzing HyDPD-GA's capabilities:

RQ4. *Can HyDPD-GA detect more abstract architectural patterns, in particular the Model-View-Controller (MVC) pattern?*

RQ5. *How does HyDPD-GA perform against the 23 GoF patterns and in comparison to related work?*

RQ6. *How does HyDPD-GA and HyDPD-ML perform against a large project in comparison to related work?*

RQ7. *What performance latency and scalability can one expect with using HyDPD-GA and how does it compare with HyDPD-ML?*

The evaluation consisted of seven parts, each addressing a research question: A) HyDPD-ML with three common design patterns, B) HyDPD-GA with three common design patterns, C) hybrid HyDPD with the same patterns, D) HyDPD-GA to probe its ability with an architecture pattern: MVC, E) HyDPD-GA across all 23 GoF patterns, and F) HyDPD-GA performance latency and scalability

The software configuration used in the evaluation consisted of srcML v1.0, Python 3.8, MongoDB v4, and Neo4j 4.2. Python libraries included: simplejson 3.17.4, pymongo 3.12.0, neo4j 4.2.0, tensorflow 2.6.0, googletrans 3.1.0a0, scikit-learn 0.24.2, keras 2.6.0, beautifulsoup4 4.9.3, numpy 1.19.5, matplotlib 3.4.3, conda 4.10.3, dpd 0.0.1, dpdml 0.0.1, scipy 1.7.1, pip 21.2.4, pandas 1.3.2, jupyter-client 6.1.12, jupyter-core 4.7.1, jupyter-server 1.10.2. The hardware configuration consisted of a PC with an i5-10210U@1.6GHz CPU, 8GB RAM, 1TB SSD running W10 Home. Docker v20.10.8 was used to containerize the services: jupyter-notebook, neo4j, mongo, and mongo-gui.

Dirty datasets: While it would be feasible to only involve both pure training and pure test datasets (manually removing or fixing all incorrectly implemented or mislabeled projects) and thus boost the accuracy numbers, we instead chose to include the real-world datasets as they were labeled by the authors of the projects we found, even though the authors may have incorrectly implemented or labeled the patterns. Thus, we intentionally include real-world impurity and our calculated accuracies reflect this, rather than achieving the 100% accuracy possible had we manually precleaned training and test datasets.

A. HyDPD-ML Evaluation (Three Patterns)

To address **RQ1** and serve as a basis for comparison, the HyDPD-ML evaluation consisted of three steps: 1) dataset acquisition, 2) supervised training, and 3) testing.

1) Dataset Acquisition

It remains a challenge to procure sufficient code projects with implemented pattern datasets in different programming languages and various patterns for both training and testing an ANN. Due to resource constraints, we thus focused on three common patterns from each of the major pattern categories: from the creational patterns, Factory; from the structural category, Adapter; and from the behavioral patterns, Observer. We then found 25 unique single-pattern code projects per pattern small single-pattern code projects from public repositories, 49 in Java and 26 in C# (mostly from github and the rest from pattern book sites, MSDN, etc.), evenly distributed into as shown in Figure 6. They were specifically labeled as examples of these patterns and manually verified. These popular programming languages are supported by srcML, and the mix of languages permits us to demonstrate the programming language independent principle. Language inequalities between available pattern examples is likely attributable to the popularity and longevity of a language and interest in patterns in that community.

Training data: Applying hold-out validation, of the 75 projects available, we selected 60 (20 per pattern category) for training the ANN, with between 60-75% of the code projects being in Java (green) and the remainder in C# (blue) as shown in the upper section of Figure 6.

Test data: The remaining 15 projects of the 75 total (five per pattern category with three in Java and two in C#) were used for the test dataset. In order to test whether signal word pattern matching significantly impacts the ANN results, these projects were duplicated and their signal words removed or renamed, resulting in six Java (orange) and four C# (purple) projects per pattern/category as shown in the lower section of Figure 6. This resulted in 10 test projects per pattern or 30 total test projects.

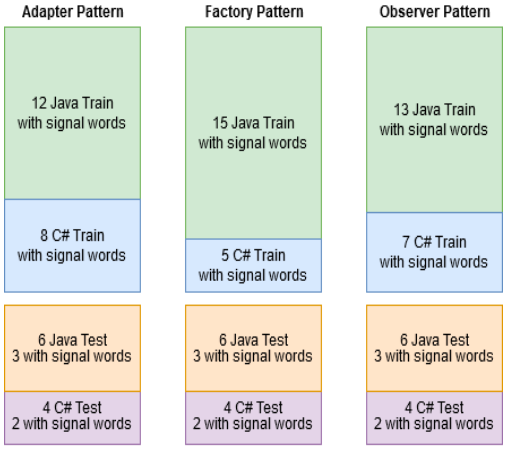

Figure 6. Pattern-specific datasets in columns with programming language specific training sets on the top rows and test sets on the bottom.

2) Supervised Training

As shown in Figure 7, during training the accuracy improves from 47% to 95% in the first seven epochs, thereafter fluctuating between 85-95% with a peak of 96.7% in the 27th epoch. The network loss metrics are shown in Figure 7. The loss value drops from an initial 1.0841 to 0.2816 in epoch 17 before small fluctuations begin, with the trend continuing downward. The loss value of 0.1995 in epoch 27 is an adequate prerequisite for detecting patterns in unknown code projects, and we saw little value in increasing the training epochs. The early stopping callback was not triggered since the overall accuracy of the network is still increasing despite the fluctuations, indicating a positive learning behavior and implying that with the given data points, it is finding structures and values that allow it to differentiate the three design patterns from each other. We thus chose to stop the training at 30 epochs, which took 2-45 seconds depending on the underlying hardware environment (any Graphical Processing Unit (GPU) with CUDA support will improve processing times).

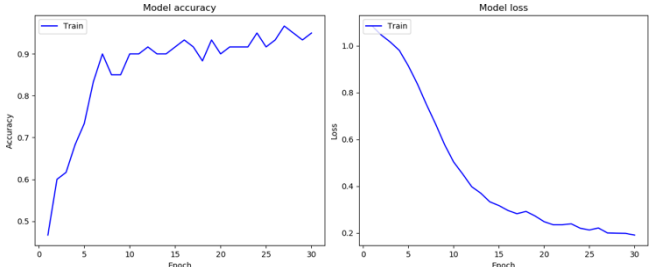

Figure 7. Network accuracy and loss over 30 epochs of training.

Considering that the worst case of random guessing would result in an accuracy of 33%, the accuracy result of 97% is significantly better and shows the potential of the approach.

The training results show that not only is the ANN learning to differentiate the patterns, its confidence for these determinations increases during the training. By epoch 27 with an accuracy of 96.7% and a loss of 0.1995, only two out of the 60 total training projects spread evenly across the three design patterns are incorrectly classified.

3) Testing

For the test dataset, 15 unique code projects were selected (five unique projects per pattern), and these were then duplicated and their signal words removed, resulting in 30 code projects. By removing the signal words, we can determine the degree of dependence of the network on these signal words.

During testing, the reported accuracy dropped to 83.3%, meaning 25 of the 30 patterns were correctly identified. Furthermore, the loss went to 0.4060, meaning a loss in confidence of its determination. A deterioration in these values is to be expected when working with unfamiliar data.

The resulting confusion matrix is shown in Table III, showing that the network was able to use its learned knowledge in training to correctly classify a majority of unknown projects (25 out of the 30 test projects). The precision column indicates how many of the predicted labels are correct, while the recall row indicates how many true labels were predicted correctly. Fewer false positives (FPs) improve the precision, while fewer false negatives (FNs) improve the recall value. All the code projects predicted to be Factory were correct (a precision of 100%), while the remaining 30% of the Factory pattern projects were incorrectly classified as another pattern, resulting in a recall of 0.70. This indicates that the Factory is more easily confused with the other patterns, a possible explanation being that the metrics we used may better differentiate more involved (i.e., more complex) patterns. The other patterns had less precision (0.81 or 0.75), but a better recall of 0.90. The overall accuracy is 88.9% with an F_1 score of 0.83. In one Observer testcase, HyDPD-ML was evenly split with Factory Method (0.46 vs. 0.46) and thus categorized as a FP.

As to the influence of signal words, our hypothesis that signal words would improve the results proved hitherto unfounded. The classification precision was not affected by signal words, with 12 projects with signal words and 13 without being correctly classified. Additional test runs showed similar results (+/- one project).

TABLE III. CONFUSION MATRIX: ML TEST 10 PROJECTS PER PATTERN

Predicted Labels	**True Labels**			**Accur.**	**Precision**	**Recall**	**F_1 Score**
	Factory	***Adapter***	***Observer***				
Factory	7	0	0	90.0%	1.00	0.70	0.82
Adapter	1	9	1	90.0%	0.81	0.90	0.86
Observer	2	1	9	86.7%	0.75	0.90	0.82
Overall				88.9%	0.83	0.83	0.83

Accur. = Accuracy

TABLE IV. CONFUSION MATRIX: ML CROSS-TESTING 90 PROJECTS

Predicted Labels	**True Labels**			**Accur.**	**Precision**	**Recall**	**F_1 Score**
	Factory	***Adapter***	***Observer***				
Factory	24	0	1	92.2%	0.96	0.80	0.87
Adapter	0	24	0	93.3%	1.00	0.80	0.89
Observer	6	6	29	85.6%	0.71	0.97	0.82
Overall				90.7%	0.87	0.86	0.86

Accur. = Accuracy

Since HyDPD-GA requires no training set, and we will be comparing HyDPD-ML with HyDPD-GA and the combination as HyDPD, it is pragmatic to utilize the entire

dataset of 90 projects for our testing. Hence, we also applied ML across the entire dataset (including the training set) to serve as a reference for comparison and to ensure that DPD did not get worse when the training set is also used for testing. This result, which includes the training dataset, is shown in Table IV, showing that overall accuracy increased to 90.3%, with precision and recall increasing to 0.86 with an overall F_1 score of 0.86.

B. HyDPD-GA Evaluation (Three GoF patterns)

To answer **RQ2** and to be able to compare HyDPD-GA with HyDPD-ML for the three GoF patterns, we utilized the entire ML dataset (both the training and test data) as the test dataset at 30 test projects per pattern and 90 total. The results are shown in Table V-VII below, where for this section we are focused on the columns HyDPD-GA and its comparison to HyDPD-ML (the column HyDPD will be discussed in the following section). The Testcase ID is unique only within a specific pattern (for internal tracking), so the ID may reoccur within another pattern and refers to a different test case.

TABLE V. DPD COMPARISON: FACTORY PATTERN

Testcase	HyDPD-ML	HyDPD-GA	HyDPD
1	1.00	**0.00**	0.50
10	1.00	0.70	0.86
11	1.00	0.70	0.86
12	1.00	1.00	1.00
13	0.98	0.70	0.85
14cs	1.00	1.00	1.00
15cs	1.00	1.00	1.00
16cs	1.00	1.00	1.00
17cs	0.99	**0.00**	**0.49**
18	1.00	1.00	1.00
19cs	1.00	1.00	1.00
2	1.00	1.00	1.00
20	0.99	1.00	1.00
21	1.00	0.70	0.86
22	**0.04**	1.00	0.52
23	0.98	1.00	0.99
24cs	0.99	1.00	0.99
25cs	1.00	1.00	1.00
26	**0.02**	0.70	**0.37**
27	**0.03**	1.00	0.51
28	**0.02**	1.00	0.51
29cs	**0.03**	1.00	0.51
3	1.00	0.70	0.86
30cs	**0.05**	1.00	0.53
4	0.99	0.70	0.85
5	1.00	0.70	0.86
6	0.67	0.70	0.69
7	1.00	0.70	0.86
8	0.86	0.70	0.78
9	0.99	0.70	0.85
FN*	6	2	2

*False Negatives marked in **bold** above

In applying GA to the test datasets, certain testcases returned less than the ideal value of 1.0 (e.g., 0.75 would indicate a partial match and 0 no match). Since GA works differently than ML and can identify a specific node involved in a pattern, we can utilize the results to analyze the cause. A manual analysis found the following explanations for the discrepancies (non 1.0 values) in the HyDPD-GA column in Tables V-VII:

TABLE VI. DPD COMPARISON: ADAPTER PATTERN

Testcase	HyDPD-ML	HyDPD-GA	HyDPD
1	**0.06**	0.75	**0.40**
10	1.00	1.00	0.87
11	1.00	1.00	1.00
12	1.00	1.00	1.00
13cs	1.00	1.00	1.00
14cs	1.00	1.00	1.00
15cs	1.00	0.75	0.87
16cs	1.00	1.00	1.00
17cs	1.00	1.00	1.00
18cs	1.00	0.75	0.87
19cs	1.00	1.00	1.00
2	1.00	**0.00**	0.50
20cs	1.00	1.00	1.00
21cs	1.00	1.00	1.00
22cs	0.89	1.00	0.95
23	1.00	1.00	1.00
24	1.00	1.00	1.00
25	0.71	1.00	0.86
26cs	**0.07**	1.00	0.54
27cs	**0.04**	1.00	0.52
28	**0.05**	1.00	0.52
29	**0.06**	1.00	0.53
3	1.00	1.00	1.00
30	**0.04**	1.00	0.52
4	0.64	**0.00**	**0.32**
5	1.00	1.00	1.00
6	1.00	1.00	1.00
7	1.00	**0.00**	0.50
8	1.00	1.00	1.00
9	1.00	1.00	1.00
FN*	6	3	2

* False Negatives marked in **bold** above

Factory pattern: 1) missing either an abstract Factory, an abstract Factory Method, or both, 2) using Builder and functional programming, 3) Factory Method does not return a Product (offers a getter call to retrieve it).

Adapter pattern: 1) client missing or call of target method missing, i.e., code just shows a possible implementation without invoking the pattern 2) Adaptee calls the Adapter's method – but an Adaptee should not have to know about the adaptation (GA returns 0) 3) two different methods are used, one method overrides the target method and another method calls the Adaptee's method, 4) the Adapter does not actually call the Adaptee (GA returns 0).

Observer pattern: 1) missing abstract Subject or abstract methods for notification, 2) missing methods to add or remove Observers from the internal list, 3) an external iterator, events, or delegation was used (GA returns 0).

The results show no overlap of FNs occurred across all 90 test cases - an indication of how each component differs and how they can be used together to complement one another. HyDPD-GA performed as good or better than HyDPD-ML for each pattern with the exception of the Observer pattern with 2 FNs (False Negatives) versus 1 FN for HyDPD-ML.

Table VIII shows the result for cross-testing the three patterns across the 90 test cases resulting in 270 tests in total (TN = true negative). HyDPD-GA had a total of 7 FNs and 1

FP across the testcases, which compares well against the 13 FNs and 13 FPs for HyDPD-ML. Overall HyDPD-GA shows as good or better recall, precision, accuracy, and F_1 scores than HyDPD-ML, with the exception of the single FP for the Adapter pattern resulting in 96.4% vs. 100% precision, and the additional FN in the Observer pattern resulting in a recall of 93.3% vs. 96.7%.

TABLE VII. DPD COMPARISON: OBSERVER PATTERN

Testcase	HyDPD-ML	HyDPD-GA	HyDPD
1	1.00	0.70	0.85
10	1.00	1.00	1.00
11	1.00	1.00	1.00
12	1.00	1.00	1.00
13cs	0.99	**0.00**	**0.49**
14cs	1.00	1.00	1.00
15cs	1.00	1.00	1.00
16cs	1.00	1.00	1.00
17	1.00	0.70	0.85
18cs	1.00	**0.00**	0.50
19cs	1.00	1.00	1.00
2	1.00	1.00	1.00
20cs	1.00	1.00	1.00
21cs	1.00	1.00	1.00
22cs	0.95	1.00	0.97
23	1.00	1.00	1.00
24	0.95	1.00	0.97
25	1.00	1.00	1.00
26cs	0.91	1.00	0.95
27cs	0.95	1.00	0.97
28	0.94	1.00	0.97
29	0.95	1.00	0.97
3	**0.46**	0.70	0.58
30	0.94	1.00	0.97
4	1.00	1.00	1.00
5	1.00	0.70	0.85
6	1.00	1.00	1.00
7	1.00	1.00	1.00
8	1.00	1.00	1.00
9	1.00	0.70	0.85
FN*	1	2	1

* False Negatives marked in **bold** above

C. HyDPD Evaluation (Three GoF Patterns)

To answer **RQ3**, for all three patterns corresponding to Table V through Table VII, the hybrid probability (1) was calculated from the HyDPD-ML and HyDPD-GA results, with the result shown in column HyDPD. Across all three patterns, every single FN from either of the techniques was compensated by a partial or full detection by the other technique, with 0.32 for Adapter testcase 4 being the lowest combined score. the combination often compensates for a FN from another component as can be seen in the tables with the bold FNs. Furthermore, in practical use perhaps a threshold such as 0.3 instead of 0.5 could be used to trigger detection.

Thus, the resulting combination as HyDPD provides a recall as good or better than any single component. Thus, HyDPD improves DPD by compensating for a FN of any isolated HyDPD-ML or HyDPD-GA value, since one may not detect a pattern that the other component can. While this may result in more FPs, we are of the opinion that the benefits of automation improve efficiency sufficiently that one would rather manually quickly verify a detection as false (FP) rather than misleading FNs. Thus, we prefer to minimize the miss rate or false negative rate (FNR).

TABLE VIII. 270 CROSS-TEST DPD SUMMARY

Component	Result	Factory	Adapter	Observer	Total
HyDPD-ML	FP	0	0	12	13
	FN	6	6	1	13
	TP	24	24	29	77
	TN	60	60	48	167
	Recall	80.0%	80.0%	96.7%	85.6%
	Precision	100.0%	100.0%	70.7%	85.6%
	Accuracy	93.3%	93.3%	85.6%	90.4%
	F_1 Score	88.9%	88.9%	81.7%	85.6%
HyDPD-GA	FP	0	1	0	1
	FN	2	3	2	7
	TP	28	27	28	83
	TN	60	59	60	179
	Recall	93.3%	90.0%	93.3%	92.2%
	Precision	100.0%	96.4%	100.0%	98.8%
	Accuracy	97.8%	95.6%	97.8%	97.0%
	F_1 Score	96.6%	93.1%	96.6%	95.4%
HyDPD	FP	0	1	0	1
	FN	2	2	1	5
	TP	28	28	29	85
	TN	60	59	60	179
	Recall	93.3%	93.3%	96.7%	94.4%
	Precision	100.0%	96.6%	100.0%	98.8%
	Accuracy	97.8%	96.7%	98.9%	97.8%
	F_1 Score	96.6%	94.9%	98.3%	96.6%

HyDPD results in Table VIII show improved results, with only 5 FNs and 1 FP out of the 270 test cases, resulting in 94.4% recall, 98.8% precision, 97.8% accuracy, and an F1 score of 96.6%. In all cases, HyDPD provided as good or better results than either HyDPD-ML or HyDPD-GA alone.

D. HyDPD-GA Evaluation (MVC Architectural Pattern)

With regard to **RQ4**, 20 MVC test pattern examples (16 in Java and 4 in C#) were acquired and HyDPD-GA applied. The results were 16 TPs, 4 FNs, and 10 FPs. Recall is 0.80, precision 0.62, accuracy 0.53, with an F_1 score of 0.70. The FNs were due to alternative implementations that deviated from the formal pattern expectation, while the high number of FPs was due to keeping the Cypher query abstract in order to maximize DPD given the numerous possibilities the architectural pattern could be implemented. We are of the opinion that we would rather verify a positive and determine a FP than miss a DPD due to a FN. Thus, we prefer to minimize the miss rate or FNR. Despite the worse results in comparison to the three GoF patterns, we believe HyDPD-GA to be a potentially promising technique for architectural pattern detection as well, and intend to investigate this further in future work.

E. Evaluation of HyDPD-GA with all GoF Patterns

RQ5 focuses on the 23 GoF patterns. Due to resource and time constraints, it was not feasible to train and evaluate the HyDPD-ML component (both alone and in conjunction with GA) against all remaining 20 GoF patterns at 25 sample projects per pattern, which would require the manual acquisition of an additional 500 code project samples. As

HyDPD-GA performed well with relatively good accuracy for the three patterns evaluated previously in sections B and C, and since it requires no training sets, our GoF evaluation only utilized HyDPD-GA. For the remaining 20 GoF patterns, from GitHub and further sources we acquired at least 6 pattern examples (3 in Java und 3 in C#) per GoF design pattern as a test dataset.

1) Testdata

The Cypher query for each pattern was applied to its own pattern test data and tuned as described in the previous section IV.B to maximize its TP and TN. Then the queries were applied to the entire GoF pattern test set consisting of 258 tests. The cross-testing resulted in 5934 tests being executed. A result of 1 was treated as positive, 0 negative, and in-between values manually analyzed. Table IX shows the GoF DPD results, where X stands for Cross-pattern detection and indicates the number of unexpected detections of that pattern in a different pattern test set. These deviations were then manually analyzed to determine if that pattern did indeed occur in the other test set or if it was a FP, shown in the corresponding column.

TABLE IX. HYDPD-GA GOF DPD.

	TC	FN	X	FP	TP	TN	A	P	R	F_1
Abstract Factory	6	0	1	0	7	251	1.00	1.00	1.00	1.00
Builder	9	0	2	2	9	247	0.99	0.82	1.00	0.90
Factory Method	40	2	26	2	62	192	0.98	0.97	0.97	0.97
Prototype	12	1	0	0	11	246	1.00	1.00	0.92	0.96
Singleton	8	0	0	0	8	250	1.00	1.00	1.00	1.00
Adapter	33	3	15	15	30	210	0.93	0.67	0.91	0.77
Bridge	7	0	1	0	8	250	1.00	1.00	1.00	1.00
Composite	10	0	10	0	20	238	1.00	1.00	1.00	1.00
Decorator	14	0	6	6	14	238	0.98	0.70	1.00	0.82
Façade	6	1	1	1	5	251	0.99	0.83	0.83	0.83
Flyweight	14	1	0	0	13	244	1.00	1.00	0.93	0.96
Proxy	6	1	6	6	5	246	0.97	0.45	0.83	0.59
CoR	6	0	0	0	6	252	1.00	1.00	1.00	1.00
Command	6	0	1	0	7	251	1.00	1.00	1.00	1.00
Interpreter	6	1	3	3	5	249	0.98	0.63	0.83	0.71
Iterator	6	1	0	0	5	252	1.00	1.00	0.83	0.91
Mediator	6	1	0	0	5	252	1.00	1.00	0.83	0.91
Memento	6	0	0	0	6	252	1.00	1.00	1.00	1.00
Observer	30	3	3	3	27	225	0.98	0.90	0.90	0.90
State	7	0	5	5	7	246	0.98	0.58	1.00	0.74
Strategy	6	1	6	3	8	246	0.98	0.73	0.89	0.80
Template Method	7	0	5	0	12	246	1.00	1.00	1.00	1.00
Visitor	7	1	0	0	6	251	1.00	1.00	0.86	0.92
Total	258	17	91	46	286	5585	0.99	0.86	0.94	0.90

X = Cross-pattern detection; A=Accuracy; P=Precision; R=Recall

A brief explanation of the FNs and FPs in Table IX:

Builder: FPs were detected in Memento, whereby an Originator instantiates the Memento object based on its own state, resulting in similar behavior.

Factory, *Adapter*, and *Observer*: the FNs are described above in Section B. One FP each in Composite and Façade.

Prototype: FN: a clone method calls a Dictionary object, resulting in an incomplete graph mapping.

Decorator: FPs: the DPD confusion occurs since Decorator, Adapter, Proxy, Interpreter, and State have structural similarities and primarily behavioral differences or differences of intent. Also, the main participant inherits functionalities from an abstract interface and has a reference to an object with this interface.

Façade: FN: the Cypher query required that the Façade class use at least 3 independent classes, but the test case uses an inheritance hierarchy, an atypical realization of the pattern.

Flyweight: FN: missing abstract Flyweight class.

Proxy: FN: the Proxy class inherits the Service, a non-compliant pattern. FPs: see Decorator explanation.

Interpreter: FN: Interpreter method does not use the Context object for accumulating results. FPs: see Decorator explanation.

Iterator: FN: an external class was used as an abstract iterator; thus the overwriting of the abstract method could not be detected.

Mediator: FN: using functional programming; separating Listener and Handler classes rather than a common class for both purposes.

State: FPs: see Decorator explanation.

Strategy: FN: the client does not create specific strategies and does not define which strategy to use; decision made in the Context class, which does not reflect the classic pattern definition.

Visitor: FN: The Visitor methods, which apply different logic depending on the type of argument, are missing; there is only one method with the type of the parent class. This violates the pattern definition.

To summarize, out of 258 testcases there were 286 TPs, 17 FNs, 46 FPs, and 5585 TNs, resulting in 0.99 accuracy, 0.86 precision, 0.94 recall, and an F_1 score of 0.90. We believe this rate to be relatively good for application on this real-world sampling. 91 cross detections triggered a manual analysis with half of them being FPs. Due to their similarities, certain patterns remain challenging to differentiate based only on GA. Note that, despite being labeled as such on the internet, a number of the FNs were non-compliant or atypical implementations, affecting the accuracy rate. While these could have been culled beforehand, we wanted to utilize a dataset with real-world labeling. Having a larger benchmark dataset prepared or approved by experts in the future would be helpful for tuning. We note that the four lowest F_1 scores are for Proxy, Interpreter, State, and Adapter.

F. HyDPD-GA Evaluation (JUnit)

To evaluate DPD for a larger project, for **RQ6** we analyzed the latest version of JUnit source code version 5.8. The 1170 Java source files contained 83595 NCLOC (non-commented LOC) out of 143440 total lines. Since we do not presume to be familiar with the architecture of JUnit, we used a manual case-independent partial keyword search that included all the signal words used to train HyDPD-ML as well as certain other terms the GoF book contains with that pattern, including "also known as" or component or method names.

The results for the three GoF patterns to compare HyDPD-GA and HyDPD-ML are shown in Table X. Since the term "factorymethod" was found 137 times in 21 files, the range of HyDPD-GA and HyDPD-ML results seem probable. As adapter-related terms also occur relatively frequently, the range of results for HyDPD-GA and HyDPD-ML also seem probable. For Observer, since HyDPD-ML had the worst F_1

score with a precision of only 70.7%, in our opinion the high number of 689 hits are unlikely related to an actual implementation of the pattern and we would tend to see the HyDPD-GA results as more likely. If proved empirically true by further large project testing, one could, for example, tailor the HyDPD weightings of (1) for the Observer pattern more heavily towards HyDPD-GA.

TABLE X. DPD COMPARISON FOR JUNIT (THREE GOF PATTERNS)

Pattern	HyDPD-GA Hits [p<1]	HyDPD-ML Hits	Lexical Search	
			Keyword*	***Hits(Files) [raw]***
Factory Method	33 [24@0.71]	150	*factory*	972(185)
			create	1099(202)
			implements	417(267)
			type	3455(367)
			factorymethod	137(21)
Adapter	102 [13@0.75]	15	*adapter*	88(23)
			adaptee	8(3)
			target	538(145)
			adapt	100(24)
			wrapper	307(27)
Observer	0	689	*observer*	0
			state	204(39)
			update	0 [6(3)]
			notify	17(7)
			publish	222(58)
			subscribe	0
			subject	0
			attach	0 [3(2)]
			detach	0
			register	706(109)
			unregister	14(4)
			deregister	0
			setstate	0
			getstate	0

* partial any case code search; ML signal words in *italics* **[raw] values revised where obvious

For small results where it was obvious, raw values sometimes were adjusted if the search result context made it clear the pattern was not involved, e.g., the use of the word outside of a pattern context in a comment or in error handling code for a different purpose. This was in no way a systematic analysis of each search result.

HyDPD-GA was used for checking all 23 GoF patterns and provided various pattern detections that lexical analysis also found indicators for. Table XI compares our results with those of other work we found that published GoF DPD for JUnit, which utilized much older versions of JUnit. Nevertheless, we can compare the reported results by GoF pattern to see the relative extent of detection for such a project. Additionally, we performed a lexical search of JUnit v5.8 to determine if the pattern name as a keyword indicates possible usage, and this was marked next to our HyDPD-GA numerical result. While related work used only older JUnit versions, in comparison it does not seem to be completely off track in the detections, except for the high number of Adapter detections. As an explanation, for the three GoF patterns, HyDPD-GA had its lowest DPD scores for the Adapter and for all GoF its precision was 0.67 with an F_1 score of 0.77. Thus, the Adapter pattern is possibly confused and not necessarily as high, yet the lexical results in Table X may make the higher number possible relative to what related work had found.

TABLE XI. FULL GOF DPD COMPARISON FOR JUNIT

	HyDPD-GA	**Dwivedi** [28]	**Mayvan** [17]	**Oruc** [19]		**Yu** [16]	**nrp** [34]	**Sempatrec** [43]	**SSA** [44]
Year	2022	2018	2017	2016		2015	2014	2014	2006
Version	v5.8	-	v3.8, v4.1	v3.8	v4.1	v3.8		v3.7	v3.7
Abstract Factory	0*	6	0			0	0	0	na
Builder	11*					0			
Factory Method	33*		1			2	0	0	0
Prototype	1					0			
Singleton	7*		0	0	4	0	0	0	0
Adapter	102*	11	4			9	6	1	6
Bridge	3*	9		2	4	0			
Composite	0*		1	1	2	0	1	1	1
Decorator	1*		1	1	1	2	1	1	1
Façade	**					0			
Flyweight	1					0			
Proxy	1*					0			
CoR	0*					0			
Command	3*					0			
Interpreter	3					0			
Iterator	8*					0			
Mediator	1					0			
Memento	0					0			
Observer	0		3			1	3	1	1
State	0*		3			0	3	4	3
Strategy	0*					0			
Template Method	8*	38	1	12	22	1	1	1	1
Visitor	0*		0			0	0	0	0

*Manual lexical search indicates possible usage (may just use/extend Java API) **Memory issue

The results indicate that HyDPD-GA can be utilized on a larger project and potentially find or detect patterns. As the HyDPD-GA accuracy rates for GoF as shown in Section E above were relatively good, we expect the results for JUnit to be comparable. However, we note the issues mentioned in that previous section, where similar patterns that are mostly differentiated by intention can result in a different labeling to a similar pattern (e.g., Decorator, Adapter, Proxy, Interpreter, and State being similar in structure), being thus more easily confused and having lower F_1 scores. Detection would require a more in-depth analysis to determine if there are issues.

G. HyDPD Performance Evaluation

DPD performance was measured as depicted in Table XII for small projects (50 to 400 LOC from the test data sets) as well as for JUnit 5.8 (to exemplify a large project). The values are depicted on a log scale in Figure 8. The differences in latency are due to the varying number of positive (required) elements (nodes and relations) that need to be matched in a Cypher query while ensuring that negative unwanted elements are not in the structure. The queries thus vary in complexity and in turn affect latency. For instance, Interpreter has many conditions as well as negative conditions, whereas Singleton requires one class as a participant and has no negative conditions. The effects become more noticeable when analyzing larger projects.

TABLE XII. HYDPD-GA LATENCY

Pattern	Small project average (seconds)	JUnit 5.8 (seconds)
Abstract Factory	0.04	0.09
Builder	0.02	9.77
Factory Method	0.02	0.10
Prototype	0.02	0.21
Singleton	0.02	0.05
Adapter	0.04	183.71
Bridge	0.04	0.26
Composite	0.02	0.04
Decorator	0.02	23.81
Façade	0.02	error
Flyweight	0.02	1.41
Proxy	0.03	0.14
CoR	0.08	1.90
Command	0.03	12.60
Interpreter	0.02	692.98
Iterator	0.02	0.23
Mediator	0.02	0.07
Memento	0.03	1.59
Observer	0.03	3.23
State	0.02	3.01
Strategy	0.05	58.34
Template Method	0.02	0.21
Visitor	0.02	0.31
Total	0.63	994.06
Average	0.03	43.22

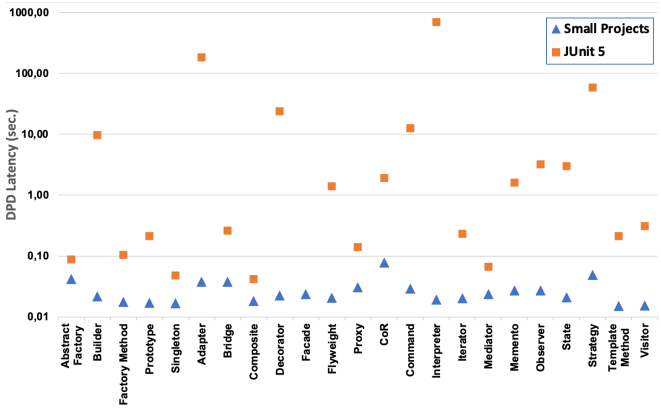

Figure 8. HyDPD-GA per-pattern latency: small project average vs. JUnit (log scale).

The total processing time needed for conversion, import, and DPD was measured as depicted in Table XIII. As one might expect for larger code bases, preparation processing time plays a more significant role, notably vector conversion for HyDPD-ML and Neo4j import for HyDPD-GA. During DPD execution, however, HyDPD-ML is not significantly impacted in contrast to HyDPD-GA.

To address performance for larger projects, one workaround might be to apply HyDPD-GA selectively for only certain pattern searches, or to apply HyDPD-ML initially since it executes much more quickly, and then selectively apply HyDPD-GA to certain patterns or only to certain modules to confirm those that HyDPD-ML detected.

TABLE XIII. TOTAL PROCESSING LATENCY

Process step	Small project average (sec.)		JUnit 5.8 (sec.)	
	ML	*GA*	*ML*	*GA*
scrML conversion	0.11	0.10	29.2	29.24
Training	5.90		5.90	
Vector conversion	16.75		22385.25	
MongoDB import		0.01		1.91
Neo4j import		0.08		948.48
Total preparation	22.74	0.19	22420.36	979.62
DPD execution	0.01	0.63	0.11	994.06
Total	22.75	0.82	22420.47	1973.69

H. Evaluation Discussion

The discussion of the evaluation results follows the **RQ**s:

RQ1: HyDPD-ML did demonstrate its feasibility, showing practical DPD results in cross-testing three common GoF patterns, with overall 90.7% accuracy, 87% precision, 86% recall, and an F1 score of 0.86.

RQ2: HyDPD-GA showed its feasibility, and performed relatively well against the three common GoF patterns, finding fewer but different overall fewer FNs and FPs than HyDPD-ML. Overall HyDPD-GA shows as good or better recall, precision, accuracy, and F_1 scores than HyDPD-ML, with the exception of the single FP for the Adapter pattern resulting in 96.4% vs. 100% precision, and the additional FN in the Observer pattern resulting in a recall of 93.3% vs. 96.7%.

RQ3: For the three common GoF patterns, the hybrid HyDPD demonstrated its feasibility, performing well with results of 94.4% recall, 98.8% precision, 97.8% accuracy, and an F1 score of 96.6%. In all cases, HyDPD provided as good or better results than either HyDPD-ML or HyDPD-GA alone.

RQ4: While HyDPD-GA can be useful for detecting more abstract architectural patterns, these are more challenging for GA to reliably detect due to their more abstract nature, enabling various implementation strategies. Testing the MVC pattern resulted in 0.80 recall, 0.62 precision, 0.53 accuracy, and an F1 score of 0.70.

RQ5: For checking HyDPD-GA against all 23 GoF patterns, cross-testing our 258 GoF testcases resulted in 5934 tests. It performed well, providing 0.99 accuracy, 0.86 precision, 0.94 recall, and an F_1 score of 0.90. It thus appears to provide quite useable results by itself. This could be especially suitable when larger datasets necessary for training (which HyDPD-ML would require) are unavailable.

RQ6: Based on the relatively large project JUnit (83K NCLOC), when comparing HyDPD-GA to HyDPD-ML for the three GoF patterns, a range difference in hits was observed, which correlates with our previous analysis that FNs of one component are often compensated as TPs by the other, or in other words, one DPD technique is better than another in certain circumstances. A lexical analysis of the code provided insights into the likelihood of the pattern usage, and the low precision of 70.7% for HyDPD-ML for the Observer pattern and the lack of clear lexical evidence would indicate it has a high FP rate for this pattern. HyDPD-GA performed relatively well. HyDPD-GA was used for checking all 23 GoF patterns and provided various pattern detections

that lexical analysis also found indicators for. While related work used only older JUnit versions, in comparison it does not seem to be completely off track in the detections, except for the high number of Adapter detections. As an explanation, for the three GoF patterns, HyDPD-GA had its lowest DPD scores for the Adapter and for all GoF its precision was 0.67 with an F_1 score of 0.77. Thus, the Adapter pattern is possibly confused and not necessarily as high, yet the lexical results in Table X may make the higher number possible relative to what related work had found.

RQ7: HyDPD-GA performance latency and scalability showed that for simpler queries its DPD performance in relative magnitude is on par with HyDPD-ML (for JUnit 5.8 a few seconds or less), but that particular patterns (in particular Builder, Adapter, Decorator, Command, Interpreter, Façade, Strategy) do require much longer query times. Preparation processing time plays a significant role before DPD can be executed, especially vector conversion for HyDPD-ML and Neo4j import for HyDPD-GA.

To summarize the evaluation, HyDPD has shown that it is viable for DPD for multiple programming languages. While combining the different strengths of HyDPD-ML and HyDPD-GA, HyDPD can also compensate for certain weaknesses of the other and improves the overall DPD capability (e.g., fewer FNs and improved F_1 score) while allowing for tailoring in weighting. More abstract architectural patterns such as MVC, while more challenging due to their abstract nature, can also be detected. For situations where insufficient training data is available for HyDPD-ML, HyDPD-GA can also be used alone and showed relatively good DPD results. Additionally, if performance and scalability are a primary factor, one alone can be chosen to lessen the impact on preparation or execution.

VI. Conclusion

This paper presented our HyDPD solution, a hybrid approach for generalized DPD utilizing graph analysis (GA) and Machine Learning (ML) and programming-language-agnostic approach to automate the detection of design patterns via source code analysis. Its realization demonstrates its feasibility, using srcML as a common markup language to support multiple programming languages, and its generalized approach works for many different patterns. The HyDPD-ML component was realized with TensorFlow using static code metrics and semantic analysis, while the HyDPD-GA component uses Cypher queries on the graph database Neo4j.

The evaluation compared each component and their combination for three common patterns across a set of 75 single pattern Java and C# public sample pattern projects. HyDPD-GA was also used to detect the 23 Gang of Four design patterns across 258 sample C# and Java projects as well as in a larger Java project JUnit. By applying the hybrid, HyDPD can compensate for certain weaknesses of the other component and improves the overall DPD capability (e.g., fewer FNs and improved F_1 score) while allowing for per-pattern or per-technique tailoring in probability weighting. More abstract architectural patterns such as MVC, while more challenging due to their abstract nature, were also detected. While HyDPD-ML requires sufficient initial training data for a pattern, HyDPD-GA can also be used alone without training and showed relatively good DPD results. Performance and scalability measurements showed the differences between components, which can be considered as to which technique to apply, with HyDPD-GA showing high performance-sensitivity for certain patterns due to the large number of matching and negative conditions that must be met.

Future work will investigate the inclusion of additional pattern properties and key differentiators to improve the results even further. This includes analyzing the network classification errors in more detail to further optimize the network accuracy, adding support for the remaining GoF patterns, utilizing semantic analysis with NLP capabilities on the code for additional natural languages, supporting additional programming languages such as C++. Also, we intend to evaluate pattern detection when they are intertwined with other patterns and address accuracy, performance, and scalability on large code bases. We will also investigate the detection of additional design and architectural patterns and implementation variants and integration with maintainer and developer tooling. Furthermore, to address the risk of overfitting, we intend to apply cross-validation and consider alternative classification schemes. Thereafter, we intend to do a comprehensive empirical industrial case study.

Acknowledgment

The author thanks Anna Kazakova, Florian Michel, and Christian Leistner for their assistance with the design, implementation, evaluation, and diagrams.

References

[1] R. Oberhauser, "A Machine Learning Approach Towards Automatic Software Design Pattern Recognition Across Multiple Programming Languages," Proc. of the Fifteenth International Conference on Software Engineering Advances (ICSEA 2020), IARIA XPS Press, 2020, pp. 27-32.

[2] M. Muro, S. Liu, J. Whiton, S. Kulkarni, "Digitalization and the American Workforce," Brookings Institution Metropolitan Policy Program, 2017. [Online] Available from https://www.brookings.edu/wp-content/uploads/2017/11/mpp_2017nov15_digitalization_full_report.pdf 2022.02.10

[3] C. Metz, "Google Is 2 Billion Lines of Code—And It's All in One Place." [Online] Available from http://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/ 2022.02.10

[4] [Online] Available from https://en.wikipedia.org/wiki/GitHub 2022.02.10

[5] G. Booch, "The complexity of programming models," Keynote talk at AOSD 2005, Chicago, IL, Mar. 14-18, 2005.

[6] [Online] Available from https://cybersecurityventures.com/application-security-report-2017/ 2022.02.10

[7] [Online] Available from https://web.archive.org/web/20210314184254/https://www.payscale.com/data-packages/employee-loyalty/least-loyal-employees 2022.02.10

[8] C. Jones, "The economics of software maintenance in the twenty first century". 2006. [Online] Available from https://web.archive.org/web/20160308070720/http://www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf 2022.02.10

[9] R. Minelli, A.Mocci, and M. Lanza, "I know what you did last summer: an investigation of how developers spend their time." In: Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (pp. 25-35). IEEE Press, 2015.

[10] M.J. Pacione, M. Roper, and M. Wood, "A novel software visualisation model to support software comprehension." In: Proc.. 11th Working Conference on Reverse Engineering. (pp. 70-79). IEEE, 2004.

[11] A. von Mayrhauser and A.M. Vans, "Program comprehension during software maintenance and evolution," Computer, 28(8), pp. 44-55, 1995.

[12] E. Gamma, Design patterns: elements of reusable object-oriented software. Pearson Education India, 1995.

[13] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-oriented software architecture: a system of patterns, Vol. 1. John Wiley & Sons, 2008.

[14] L. Prechelt, B. Unger-Lamprecht, M. Philippsen and W. F. Tichy, "Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance," in IEEE Transactions on Software Engineering, vol. 28, no. 6, pp. 595-606, June 2002, doi: 10.1109/TSE.2002.1010061.

[15] M.G. Al-Obeidallah, M. Petridis, and S. Kapetanakis, "A survey on design pattern detection approaches," International Journal of Software Engineering (IJSE), 7(3), pp.41-59, 2016.

[16] D. Yu, Y. Zhang, and Z. Chen, "A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures," Journal of Systems and Software, vol. 103, pp. 1-16, 2015.

[17] B. Mayvan and A. Rasoolzadegan, "Design pattern detection based on the graph theory," Knowledge-Based Systems, vol. 120, pp. 211-225, 2017.

[18] M.L. Bernardi, M. Cimitile, and G. Di Lucca, "Design pattern detection using a DSL-driven graph matching approach," Journal of Software: Evolution and Process, 26(12), pp.1233-1266, 2014.

[19] M. Oruc, F. Akal, and H. Sever, "Detecting design patterns in object-oriented design models by using a graph mining approach," 4th International Conference in Software Engineering Research and Innovation (CONISOFT 2016), pp. 115-121, IEEE, 2016.

[20] A. Pande, M. Gupta, and A.K.Tripathi, "A new approach for detecting design patterns by graph decomposition and graph isomorphism," International Conference on Contemporary Computing, pp. 108-119, Springer, Berlin, Heidelberg, 2010.

[21] P. Pradhan, A.K. Dwivedi, and S.K. Rath, "Detection of design pattern using graph isomorphism and normalized cross correlation," Eighth International Conference on Contemporary Computing (IC3 2015), pp. 208-213, IEEE, 2015.

[22] S. Alhusain, S. Coupland, R. John, and M. Kavanagh, "Design pattern recognition by using adaptive neuro fuzzy inference system," 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, pp. 581-587, IEEE, 2013.

[23] M. Zanoni, F. A. Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," J. of Systems & Software, vol. 103, no. C, pp. 102-117, 2015.

[24] L. Galli, P. Lanzi, and D. Loiacono, "Applying data mining to extract design patterns from Unreal Tournament levels," Computational Intelligence and Games. pp. 1-8, IEEE, 2014.

[25] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, "Design pattern mining enhanced by machine learning," 21st IEEE Int'l Conf. on Softw. Maintenance (ICSM'05), IEEE, pp. 295-304, 2005.

[26] S. Uchiyama, H. Washizaki, Y. Fukazawa, and A. Kubo, "Design pattern detection using software metrics and machine learning," First International Workshop on Model-Driven Software Migration (MDSM 2011), pp. 38-47, 2011.

[27] S. Uchiyama, A. Kubo, H. Washizaki, and Y. Fukazawa, "Detecting design patterns in object-oriented program source code by using metrics and machine learning," Journal of Software Engineering and Applications, 7(12), pp. 983-998, 2014.

[28] A.K., Dwivedi, A. Tirkey, and S.K. Rath, "Software design pattern mining using classification-based techniques," Frontiers of Computer Science, 12(5), pp. 908-922, 2018.

[29] H. Thaller, L. Linsbauer, and A. Egyed, "Feature maps: A comprehensible software representation for design pattern detection," IEEE 26th international conference on software analysis, evolution and reengineering (SANER 2019), pp. 207-217, IEEE, 2019.

[30] 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," Applied Soft Computing, 26, pp. 357-367, 2015.

[31] Y. Wang, H. Guo, H. Liu, and A. Abraham, "A fuzzy matching approach for design pattern mining," J. Intelligent & Fuzzy Systems, vol. 23, nos. 2-3, pp. 53-60, 2012.

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

[33] S. Hussain, J. Keung, and A.A. Khan, "Software design patterns classification and selection using text categorization approach," Applied soft computing, 58, pp.225-244, 2017.

[34] A. Alnusair, T. Zhao, and G. Yan, "Rule-based detection of design patterns in program code," Int'l J. on Software Tools for Technology Transfer, vol. 16, no. 3, pp. 315-334, 2014.

[35] M. Lebon and V. Tzerpos, "Fine-grained design pattern detection," IEEE 36th Annual Computer Software and Applications Conference, IEEE, pp. 267-272, 2012.

[36] I. Issaoui, N. Bouassida, and H. Ben-Abdallah, "Using metric-based filtering to improve design pattern detection approaches," Innovations in Systems and Software Engineering, vol. 11, no. 1, pp. 39-53, 2015.

[37] Y. G. Guéhéneuc, J. Y. Guyomarc'h, and H. Sahraoui, "Improving design-pattern identification: a new approach and an exploratory study," Software Quality Journal, vol. 18, no. 1, pp. 145-174, 2010.

[38] F. A. Fontana, S. Maggioni, and C. Raibulet, "Understanding the relevance of micro-structures for design patterns detection," Journal of Systems and Software, vol. 84, no. 12, pp. 2334-2347, 2011.

[39] I. Issaoui, N. Bouassida, and H. Ben-Abdallah, "Using metric-based filtering to improve design pattern detection approaches. Innovations in Systems and Software Engineering," vol. 11, no. 1, pp. 39-53, 2015.

[40] J. Dong, Y. Zhao, and Y. Sun, "A matrix-based approach to recovering design patterns," IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans, vol. 39, no. 6, pp. 1271-1282, 2009.

[41] M. Collard, M. Decker, and J. Maletic, "Lightweight transformation and fact extraction with the srcML toolkit," IEEE 11th international working conference on source code analysis and manipulation, IEEE, 2011, pp. 173-184.

[42] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv preprint arXiv:1412.6980, 2014.

[43] G. Rasool and P. Mäder, "Flexible design pattern detection based on feature types," In 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 243-252, IEEE, 2011.

[44] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S.T: Halkidis, "Design pattern detection using similarity scoring," IEEE transactions on software engineering, *32*(11), pp. 896-909, 2006