A Machine Learning Approach Towards Automatic Software Design Pattern Recognition Across Multiple Programming Languages

Roy Oberhauser[0000-0002-7606-8226]
Computer Science Dept.
Aalen University
Aalen, Germany
e-mail: roy.oberhauser@hs-aalen.de

***Abstract*—As the amount of software source code increases, manual approaches for documentation or detection of software design patterns in source code become inefficient relative to the value. Furthermore, typical automatic pattern detection tools are limited to a single programming language. To address this, our Design Pattern Detection using Machine Learning (DPDML) offers a generalized and programming language agnostic approach for automated design pattern detection based on Machine Learning (ML). The focus of our evaluation was on ensuring DPDML can reasonably detect one design pattern in the structural, creational, and behavioral category for two popular programming languages (Java and C#). 60 unique Java and C# code projects were used to train the artificial neural network (ANN) and 15 projects were then used to test pattern detection. The results show the feasibility and potential for pursuing an ANN approach for automated design pattern detection.**

Keywords–software design pattern detection; machine learning; artificial neural networks; software engineering.

I. INTRODUCTION

In the area of software engineering, software design patterns have been well-documented and popularized, including the Gang of Four (GoF) [1] and POSA [2]. The application of documented solutions to recurring software design problems has been a boon to improving software design quality and efficiency.

However, as the design patterns are mostly described informally, their implementation can vary widely depending on the programming language, natural language, pattern structure and terminology awareness of the programmer, experience, and interpretation. The actual detection and documentation of these software design solution patterns has hitherto relied on the experience, recollection, and manual analysis of experts. The pattern books referenced above were published over 25 years ago, and while many millions of lines of code have since been programmed, they have not been subjected to any comprehensive analysis. Furthermore, any project documentation of applied patterns, if existent, may be inconsistent with the current source code reality (e.g., prescriptive documentation of intentions, adaptations during development, maintenance changes) and thus not reflected or necessarily trustworthy. Additionally, known pattern variants may occur, the patterns may evolve over time with technology, and in fact new patterns may unknowingly be developed that the experts may be unaware of. The many different programming languages used, the different natural languages of programmers that affect naming, tribal community effects, the programmer's (lack of) knowledge of these patterns and use of (proper) naming and notation or markers, make it difficult to identify pattern usage by experts or tooling. While many code repositories are accessible to the public on the web, many more repositories are hidden within companies or other organizations and are not necessarily accessible for analysis. While determining actual pattern usage is beneficial for identifying which patterns are used where and can help avoid unintended pattern degradation and associated technical debt and quality issues, the investment necessary for manual pattern extraction, recovery, and archeology is not economically viable.

While automated feature extraction of software design patterns from documentation or code repositories is not yet commonly available among popular software development tools, research has attempted to find automated techniques that work. However, most of the published techniques have not applied ML to this problem area. One implicit challenge for most approaches is to demonstrate coverage of all 23 of the GoF patterns, which very few if any achieve.

This paper contributes Design Pattern Detection using Machine Learning (DPDML), a generalized and programming language independent approach for automated design pattern detection based on ML. Our realization of the core of the solution approach shows its feasibility, and an evaluation using 75 unique Java and C# code projects with three common GoF patterns for training and testing provides insights into its potential and limitations.

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. This is followed by our evaluation and then a conclusion.

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, with some utilizing a combination of styles. Many approaches include some form of structural analysis for pattern detection. Within this style, ML approaches use classification, decision tree, Artificial Neural Networks (ANNs), or support vector machines (SVMs), mapping the pattern detection problem to a learning problem. Examples include MARPLE-DPD [3], Galli et al. [4], and Ferenc et al. [5]. Wang et al. [6] uses a reason-based approach based on matrices. Examples of rule-based approaches include

 ISBN: 978-1-61208-827-3

Sempatrec [7] and FiG [8], which use an ontology representation. Metric-based approaches include MAPeD [9] and PTIDEJ [10]. Fontana et al. [11] describe a micro-structure-based structural analysis approach. In the behavioral analysis style, graph-based approaches include: DPIDT [12] that analyzes subpatterns in UML, and a UML semantic graph by Mayvan and Rasoolzadegan [13]. An example semantic-analysis style approach is Issaoui et al. [14], which uses an XML representation. DP-Miner [15] is a matrix-based approach using UML that involves structural, behavioral, and semantic analysis. Uchiyama et al. [16] uses a metric-based approach that involves both structural and behavioral analysis.

The styles and approaches 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. We conclude further investigation and research in this area is essential to enhancing the knowledge surrounding this area. Our solution approach is unique in offering: 1) a generalized code-centric approach that combines available data (rather than focusing on only one category of information) without necessarily requiring behavioral analysis, 2) being programming language-independent to support multiple popular programming languages, and 3) leveraging ML.

III. SOLUTION

Our full holistic DPDML solution approach is shown in Figure 1, indicating the realized DPDML-C core subset. It is based on the following principles:

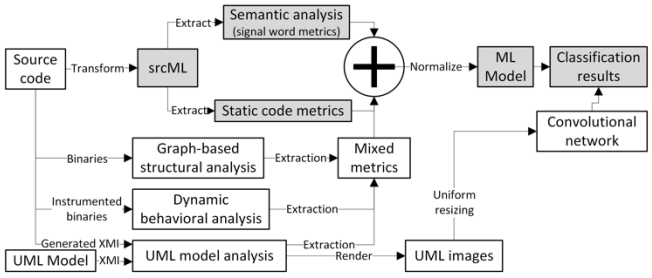

Figure 1. General DPDML comprehensive solution approach with realized core DPDML-C (shown in grey).

ML model: by utilizing ML to analyze sample data, the model learns how to classify new unknown data, in our case to differentiate design patterns. Our realization may apply or combine any ML model that suites 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.

Programming language-independent: the source code is converted into an abstracted common format for further processing. For this, in our realization we currently utilize srcML [17], thus our realization can currently support any programming languages that have a mapping to the srcML XML-based format, including C, C++, Java, and C#. If other abstract syntax formats are standardized and available for analysis, these also can be considered. Our main purpose is to extract various metrics in a common fashion from the source code.

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 metric extraction: various static code metrics are utilized to detect and differentiate design patterns.

Graph analysis: code repositories are analyzed using graph-based tools like jQAssistant and metrics extracted.

Dynamic analysis: tracing runtime code behavior can detect behavioral similarities in event sequencing, especially for the creational or behavior patterns. From these traces event and related runtime metrics can be extracted.

UML structural analysis: in case a UML model exists, the XMI structures can be analyzed and indicators extracted, such as signal words or other structural metrics. If no UML diagrams or XMI exist, they could be generated by reverse engineering UML tools and structural metrics extracted. Furthermore, a convolutional network could analyze UML images for similarities to support pattern classification.

Metric normalization: the value ranges of metrics are normalized to a scale of 0-1 to improve ANN performance.

The hypothesis driving our DPDML solution and investigation is that by utilizing all available data and more specifically metrics related to the design patterns, and feeding this input into an ANN or other ML models, we can achieve suitable classification accuracy. From a practicality standpoint, this could reduce the overall manual labor involved in identifying potential patterns, classifying them, and can potentially assist developers, maintainers, or experts involved in software archeology.

IV. (PARTIAL) REALIZATION

For our realization to apply ML, a sufficient data set of different and realistic projects was needed to support supervised learning. Not all portions of the full solution approach could yet be realized due various unexpected obstacles and project resource constraints, and we plan to address the complete DPDML realization in future work.

A. Comprehensive DPDML Challenges

UML structural analysis: most of the 60 design pattern code repositories we used did not contain any or sufficient UML for us to use in supervised training or testing. If they contained UML, it would be time-consuming to manually verify the code to determine if they are correspondingly valid UML diagrams. If they were created manually rather than generated, they may contain some additional information or

 ISBN: 978-1-61208-827-3

signal words not necessarily available in the code. If, however, round-trip UML tools were used, then the code reflects the information found in the diagrams, and thus the diagrams hold no additional information. While UML can be helpful for human analysis and verification because it distills code structure visually, they are difficult to automatically verify against code and machine-based analysis does not necessarily benefit from or need the simplification. If UML diagrams were generated directly from the code by a UML tool triggered by DPDML, little additional value would currently be gained, since the basis is the code itself, and no structural information not already contained or derivable from the code is created. Given a common UML generator, structural visual image comparison techniques could be applied with a convolutional network to determine if it helps with classification. Lacking a UML training and testing dataset of sufficient size, this portion of our solution concept will be evaluated in future work.

Dynamic analysis: many available code projects have different runtime environments, languages, libraries, concurrent processing, and require specialized tooling to acquire behavior tracing data, which is very computing resource intensive, time consuming to manually setup and acquire, and requires specialized automated analysis tools, since no formats or tool standards exist in this area. In the tracing, one would have to ensure that the patterns are actually substantially executed, which can be issue for larger projects. Furthermore, creating sufficiently large training sets for ML would be challenging. For most users of the approach we are seeking, requiring this level of analysis would perhaps be an academic exercise and could improve our understanding, but it is neither practical nor economically viable for continued usage, and we thus did not realize this portion of our solution concept.

Graph analysis: Analysis of code repositories using graph-based tools such as jQAssistant could be used, but similar to the dynamic analysis issues, tools such as jQAssistant require compiled binaries for analysis. Given this, it could be used to query various aspects and enhance our classification results and can be used to assist with manual verification. However, since our training data did not consist wholly of compiled code, we intend to address the realization of this aspect in future work.

Various analysis tools could potentially improve the results, but these are usually developed with a certain purpose that influences the interaction modes and the output. For instance, plugins for the Eclipse IDE are often focused on Java, are primarily graphical to help a developer analyze the current project, but are not designed for automated analysis of many projects in various languages from the command line. Since we chose to include both Java and C# support, no IDE-specific tooling was utilized. Beyond IDE tools, reverse-engineering tools such as Imagix 4D or code analysis tools like SourceMeter require either commercial licenses or are missing a command-line mode, and are limited in how they can be used for automated analysis situations in our context.

B. Core DPDML-C Implementation

A key aspect of our investigation was to determine if the core of the DPDML solution, metrics-based ML using an ANN, works as intended. Also, since source code should usually be readily available, whereas other information (binaries may not build, instrumentation and UML may not exist), our prototype realization effort focused on the source code analysis, known as the core DPDML-C as shown in grey in Figure 1. Due to resource and time constraints, we initially focused on having the network learn to detect one pattern out of each of three pattern categories: from the structural category, Adapter; from the creational patterns, Factory; and from the behavioral patterns, Observer. This pattern scope could then be expanded in future work if the outcome is positive.

Python was used to implement our 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.

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. [16] 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

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 variation*s: To determine if other signal words beyond the design pattern name were used in implementations, we analyzed several examples of implemented design patterns and any UML diagrams, if provided. 12 additional signal words were selected, four for each pattern as shown in Table II.

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

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.

C. Artificial Neural Network (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 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.

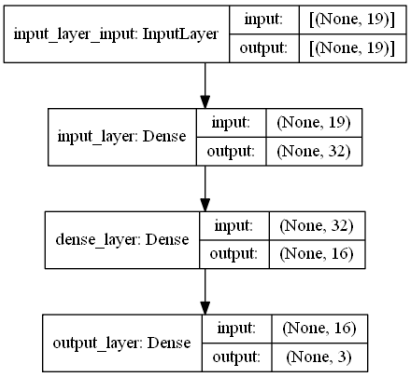

Figure 2. ANN model overview created with Keras.

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.

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 [18]. No regularization was applied in each layer. Adam automatically adjusts and optimizes the learning rate. Sparse categorical crossentropy 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.

D. 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 DPDML 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 this will be considered in future work.

V. EVALUATION

The evaluation corresponds to the three patterns that were the focus of the realization: from the structural category, Adapter; from the creational patterns, Factory; and from the behavioral patterns, Observer. The reason for choosing these three is that each represents a different pattern category and these are popular patterns. Furthermore, the number of While such simple design patterns might well be better detected with other ML models, our overall DPDML is much more ambitious, and we thus wanted to validate that an ANN would still work suitably (perhaps not optimally) in a more constrained low-data case.

 ISBN: 978-1-61208-827-3

A. Dataset

As shown in Figure 3, the dataset consisted of 75 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 25 unique code projects per pattern. They were specifically labeled as examples of these patterns, and manually verified. These popular languages are supported by srcML, and the mix permits us to demonstrate the programming language independent principle. The inequality between language examples is likely due to the language popularity and age.

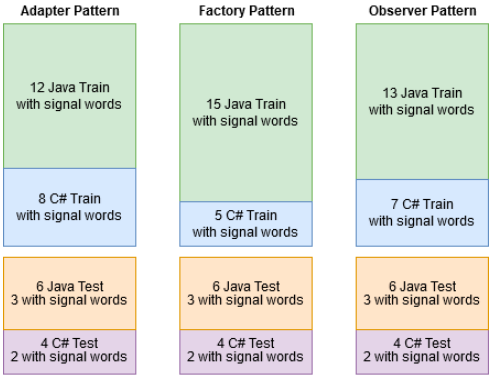

Figure 3. Pattern-specific datasets in columns with programming language specific training sets on the top rows and test sets on the bottom.

Training data: Of the 75 projects, 20 per pattern category (60 total) were selected 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 3.

Test data: The remaining 15 projects of the 75 (five per pattern category with 3 in Java and 2 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 6 Java (orange) and 4 C# (purple) projects per pattern/category as shown in the lower section of Figure 3. This resulted in 10 test projects per pattern.

B. Supervised Training

As shown in Figure 4, 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 4. 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).

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.

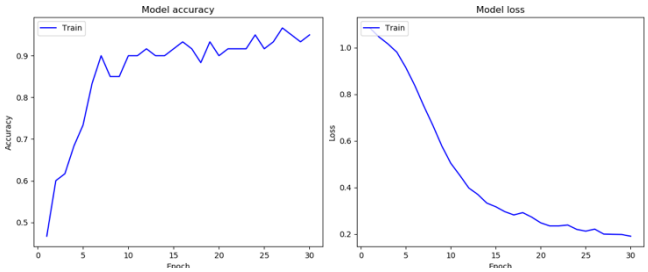

Figure 4. Network accuracy and loss over 30 epochs of training.

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 code projects spread evenly across the three design patterns are incorrectly classified.

C. Testing

Recall from Section V.A. that for the test dataset, 15 unique code projects were taken (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 results show that the network was able to use its learned knowledge in training to correctly classify a majority of unknown projects (25 out of 30).

TABLE III. CONFUSION MATRIX BASED ON 30 CODE PROJECTS

Predicted Labels	**True Labels**			**Accuracy**	**Precision**	**F_1 Score**
	Factory	***Adapter***	***Observer***			
Factory	7	0	0	90%	100%	0.82
Adapter	1	9	1	90%	81%	0.86
Observer	2	1	9	86.7%	75%	0.82
Recall	70%	90%	90%			

The confusion matrix is shown in Table III. 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 improve the precision, while fewer false negatives 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 (these false negatives result in a recall of 70%). This indicates that the Factory is more easily confused

 ISBN: 978-1-61208-827-3

with the other patterns, a possible explanation being that the metrics we used may better differentiate more involved (more complex) patterns. The other patterns had less precision (81% or 75%), but a better recall of 90%. The overall F_1 score is 0.83.

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). However, in future work we will investigate this further as we increase the statistical basis.

The results show suitable accuracy of the DPDML-C, and we believe a generalization of the DPDML approach across the GoF and further patterns to be promising.

VI. CONCLUSION

This paper presented our DPDML solution approach, a generalized and source code-based but programming language-independent approach for automated design pattern detection based on ML. Our realization of the core DPDML-C shows its feasibility for source code-based analysis. An evaluation using 60 unique Java and C# code projects for training and then 15 code projects for testing. With an accuracy of 83% and loss of 0.4060 during testing, the results show the feasibility and potential for pursuing an ANN approach for automated design pattern detection as well as some of the limitations. Furthermore, no cost-intensive behavioral analysis was involved to achieve this result. Our results for the three patterns did not show that signal words substantially improve results, indicating that other pattern characteristics can potentially suffice as indicators. While our initial focus on three fundamental patterns is obviously not of practical use yet, it shows promise for extending it to others.

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++, and extending our prototype realization to include additional code metrics, UML structural analysis (if UML is available), graph-based analysis, and dynamic behavioral analysis if traces are provided. Also, we intend to evaluate pattern detection when they are intertwined with other patterns and evaluate accuracy, performance, and practicality on large code bases. We will also investigate the detection of new design patterns and variants to the traditional patterns. Furthermore, we intend to apply cross-validation and consider alternative classification schemes such as Naïve Bayes, Decision Tree, Logistic Regression, and SVMs. Thereafter, we intend to do an empirical industrial case study.

ACKNOWLEDGMENT

The author thanks Florian Michel for his assistance with the design, implementation, evaluation, and diagrams.

REFERENCES

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

[2] 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.

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

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

[5] 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, 2005, pp. 295-304.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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), p. 38-47, 2011.

[17] 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.

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

 ISBN: 978-1-61208-827-3