Presentation of the ICSEA 2023 conference paper:

Design Pattern Detection in Code:
A Hybrid Approach Utilizing a Bayesian Network, Machine Learning with Graph Embeddings, and Micropattern Rules

Roy Oberhauser and Sandro Moser
Aalen University
Germany

IARIA

The Eighteenth International Conference on Software Engineering Advances
ICSEA 2023

Presenter: Roy Oberhauser

- Professor of Computer Science at Aalen University in Germany since 2004, teaching in the areas of software engineering.
- Worked 14 years in the software industry (Silicon Valley and Germany).
- Research interest is to leverage technologies, methods, techniques, and tools to innovate, automate, support, and improve the production and quality of software for society.

Contents

- Motivation
- Challenges
- Background
- Solution
- Implementation
- Evaluation
- Conclusion

Motivation

While the amount of program source code worldwide continues to rapidly expand, code comprehension remains a limiting productivity factor.

- Program comprehension may consume up to 70% of the software engineering effort [1].
- 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 [2].
- Code that is not correctly understood by programmers impacts quality and efficiency

Software Design Patterns (DPs) have been documented and popularized, including the Gang of Four (GoF) [3] and POSA [4].

- The application of abstracted and documented solutions to recurring software design problems has been a boon to improving software design quality, efficiency, and aiding comprehension.
- These well-known macrostructures or associated pattern terminology in code can serve as beacons to abstracted macrostructures, and as such may help identify aspects such as the author's intention or the purpose of a code segment, which, in turn, supports program comprehension

Challenges for Automated Design Pattern Detection (DPD)

- Possible benefits of automated DPD development or maintenance include:
  - Quicker comprehension of DP-related structural aspects of some software;
  - Supplementing design documentation; automatically documenting DPs;
  - Reducing dependence on unreliable or incomplete manual DP documentation;
  - Detection of inadequately implemented DPs, e.g., as unknown DPs or DP variants.
- Yet automated DPD faces challenges, including:
  - Tool support for heterogeneous programming languages, as DPs are independent of programming language;
  - Internationalization and labeling, since developers may name and comment in their natural language or any way they like;
  - Varying pattern abstraction levels, such as design vs. architectural patterns;
  - Similarities and intent differentiation, since some similar pattern structures are primarily differentiated by their intention;
  - DP localization, indicating where in code a DP was detected; and
  - Detecting variants, since each implementation is unique. While various DPD approaches have been explored [5] [6], no approach has so far achieved significant traction in practice and industry tools, and thus additional investigation into further viable approaches and improvements is warranted

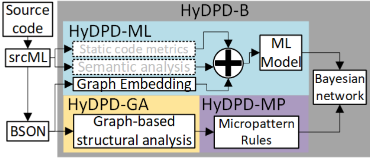
Automated DPD Approaches

- Automated DPD approaches can arguably be categorized into three primary approaches:
  - Learning-based
    - DPs are (semi-)automatically learned (e.g., via supervised learning) from provided data and requiring minimal expert intervention;
  - Knowledge-based
    - an expert defines DPs by describing elements and their associations; and
  - Similarity-based
    - DPs are grouped based on similar metrics or characteristics

Components of our HyDPD-B Solution Concept

In previous work our hybrid DPD approach (**HyDPD**) was described. **HyDPD**:

- Combines various DPD approaches.
- Converts heterogeneous source code into a common format srcML [36] for further processing by a hybrid set of subsystems:



- **HyDPD-GA**: Graph Analysis (GA) converts the srcML to BSON (Binary JSON) stored in MongoDB, maps it to a graph model stored in Neo4j that supports the Cypher Query Language (CQL) [37] for graph-based DPD analysis
- **HyDPD-ML**: Machine Learning (ML) model; for this paper uses knowledge graph embeddings as input to a supervised learning model
- **HyDPD-MP**: new MicroPatterns (MP) subsystem for expert-based approximate DP matching via MicroPattern (MP) rule catalog and Design Pattern Rule Language (DPRL) support
- **HyDPD-B**: new hybrid solution concept using a Bayesian network to integrate results from our various DPD subsystems (HyDPD-ML, HyDPD-GA, HyDPD-MP).

Improvements

HyDPD-B improvements to our previous HyDPD concept include:

- **DPRL**: Providing a mechanism to engage developers as experts in defining DP rules via a simple DP Rule Language (DPRL)
- **HyDPD-MP**: Enabling approximate DP matching via micropatterns support (HyDPD-MP)
- **HyDPD-ML**: Utilizing graph embeddings to leverage structural code graph ML results
- **Variants**: Enabling known and unknown variant detection
- **Bayesian network**: provides a *flexible* framework for probabilistic reasoning that is *comprehensible* and *interpretable* for humans, and thus offering a hybrid solution for utilizing all three DPD approaches (learning-, knowledge-, and similarity-based)

HyDPD Design Pattern Rule Language (DPRL)

- A graph-oriented rule language for developers (i.e., the knowledge experts) that should be relatively easy to learn and comprehend.
  - While the Neo4j Cypher Query Language (CQL) is powerful and offers a human-readable interface for formulating graph queries, a developer would nonetheless need to learn the Cypher syntax to formulate these only for the purpose of DPD.
  - Instead, since developers are already well acquainted with the relatively simple JSON format, we chose have DPRL conform to JSON, and then parse and map values to generate CQL.
- The primary language concepts are *participants*, *subpatterns*, and *relations*

Design Pattern Rule Language (DPRL): Adapter DP Example

- **Participants** represents a collection of participant objects in a DP.
- In its simplest form, a participant consists of the field name (line 21) – for instance, if the nature of the participant is irrelevant but the role it plays is of importance.
- The optional constraints field (line 4 and 14) allows a collection of arbitrary unary constraints (constraints that only involve the participant variable) to be specified.
- In Cypher, these constraints may correspond to labels while others may correspond to attributes. The distinction is made by our DSL parser using an internal symbol table.
- A constraint consists of three values: field (line 6 and 16) corresponding to the target of the constraint; operator (line 7 and 16) corresponding to the truth operator; and value (line 8 and 18) corresponding to the desired field value.

```
1   {
2       "participants": [
3           {
4               "constraints": [
5                   {
6                       "field": "Type",
7                       "operator": "is",
8                       "value": true
9                   }
10              ],
11              "name": "adaptee"
12          },
13          {
14              "constraints": [
15                  {
16                      "field": "Type",
17                      "operator": "is",
18                      "value": true
19                  }
20              ],
21              "name": "adapter"
22          }
23      ],
```

Design Pattern Rule Language (DPRL): Adapter DP Example

- **Subpatterns** (line 24) represents a collection of subpattern objects, each of which consists of a collection of binary relations (line 26 and 43) and the field truthvalue (line 40 and 57), indicating if the subpattern should be matched positively or negatively (precluded).
- While a pattern can contain only a single positive subpattern, it can contain an arbitrary number of negative subpatterns.

```
24            "subpatterns": [
25                {
26                    "relations": [
27                        {
28                            "constraints": [
29                                {
30                                    "field": "collection",
31                                    "operator": "is",
32                                    "value": "true"
33                                }
34                            ],
35                            "directed": true,
36                            "operand1": "adaptee",
37                            "operand2": "adapter"
38                        }
39                    ],
40                    "truthvalue": false
41                },
```

HyDPD Design Pattern Rule Language (DPRL): Adapter DP Example

- **Relations** (line 26 and 43) is a collection of relations between participants, which are specified by the fields operand1, operand2, constraints, and directed (lines 28-37).
- Operand1 and operand2 each contain either a name reference to a participant or a full description of a participant object (as described above).
- The collection constraints contains constraints analogous to those defined on a participant.

```
42                {
43                    "relations": [
44                        {
45                            "constraints": [
46                                {
47                                    "field": "collection",
48                                    "operator": "is",
49                                    "value": "false"
50                                }
51                            ],
52                            "directed": true,
53                            "operand1": "adaptee",
54                            "operand2": "adapter"
55                        }
56                    ],
57                    "truthvalue": true
58                }
59            ]
60    }
```

HyDPD Design Pattern Rule Language (DPRL): Adapter DP Example

- Our JSON DSL is automatically parsed to an equivalent Cypher query.
- For the Adapter DP example DPRL example on the right, the equivalent Cypher Query is shown below.

```
MATCH (adaptee) -[e]-> (adapter)
WHERE adaptee:Type AND adapter:Type AND e.collection = false
AND NOT EXISTS {MATCH (adaptee) -[f]-> (adapter) WHERE adaptee:Type
AND adapter:Type AND f.collection = true AND adaptee <> adapter}
AND adaptee <> adapter RETURN *
```

In our opinion, for a developer with no knowledge of Cypher, the equivalent Cypher query is more complicated to formulate or comprehend than the JSON on the right.

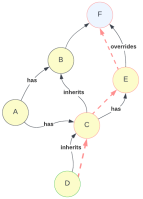
```json
{
    "participants": [
            {
                "constraints": [
                    {
                        "field": "Type",
                        "operator": "is",
                        "value": true
                    }
                ],
                "name": "adaptee"
            },
            {
                "constraints": [
                    {
                        "field": "Type",
                        "operator": "is",
                        "value": true
                    }
                ],
                "name": "adapter"
            }
        ],
        "subpatterns": [
            {
                "relations": [
                    {
                        "constraints": [
                            {
                                "field": "collection",
                                "operator": "is",
                                "value": "true"
                            }
                        ],
                        "directed": true,
                        "operand1": "adaptee",
                        "operand2": "adapter"
                    }
                ],
                "truthvalue": false
            },
            {
                "relations": [
                    {
                        "constraints": [
                            {
                                "field": "collection",
                                "operator": "is",
                                "value": "false"
                            }
                        ],
                        "directed": true,
                        "operand1": "adaptee",
                        "operand2": "adapter"
                    }
                ],
                "truthvalue": true
            }
        ]
}
```

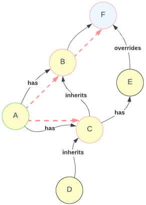
MicroPattern Catalog (MPC)

- Certain structural aspects of design patterns can ideally be expressed as a set of smaller elementary units or characteristics we refer to as MicroPatterns (MPs) [39]
  - E.g., Instantiation, Inheritance, Delegate, Extend, and Conglomeration.
- This also supports the reuse of viable MP detection components.
- Decomposing our existing graph-based queries in the Cypher Query Language (CQL) from our previous work on HyDPD-GA provided derived MPs with appropriate queries.

Randomized Graph Embeddings

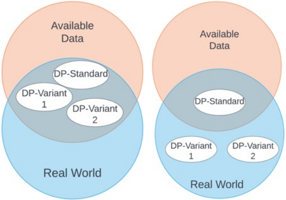
- In our previous work, HyDPD-ML was trained on tabular features extracted from source code. These features include the existence of specific semantic keywords, as well as object-oriented metrics, such as the number of classes in a project.
  - This approach is vulnerable to a change in naming convention or code obfuscation.
- To mitigate this issue, we introduce a new approach, using knowledge-graph-embeddings.
  - Input for those embeddings is provided by the graphs used by HyDPD-GA.
- We apply a simple embedding approach:
  - We first sample a predetermined number of random substructures in the graph.
  - Those substructures are always extracted from the training set to exclude possible information leakage.
  - Substructures include information about the relationship type. From those substructures, we derive a pattern query.
  - A graph embedding is created by matching all generated pattern queries against a graph. This results in binary vectors, 0 if a pattern matched, else 1.
  - While the number of generated patterns can be treated as a hyper parameter, we decided to work with 500 patterns.
  - Another hyper parameter is the complexity of extracted patterns. We define pattern complexity as the number of edge traversals in the knowledge graph (shown with complexity 3 in the figures).
  - In a grid search experiment, it was determined that constraining complexity between 3 and 4 traversals yields optimal results.
- The graph embeddings are consumed by a simple logistic regression model with L2 regularization. This enables learning from sparse data.
  - This composition of random feature extraction combined with a regularized linear model is inspired by the ROCKET-algorithm, which is used for time series classification [40]. By using a linear model, the interpretability of any results can be better supported.





Pattern Variant Detection

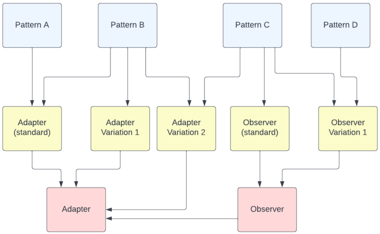
- DPs often do not conform exactly to some specification, making detection of DP variants challenging. The problem of DP variant detection can be partitioned into
  - 1) the detection of known variants, and
  - 2) the detection of unknown variants
- Assuming DP variants share a substantial degree of MPs, our solution concept should be able to detect known pattern variations efficiently.
- Moreover, by using hidden variables in the Bayesian network, the algorithm can also provide precise information regarding the variant.



Detecting known (left) and unknown (right) DP variants

Pattern Variant Detection Example

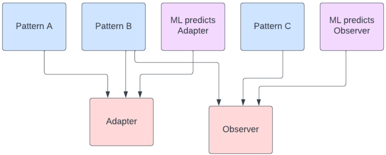
- Yellow variables correspond to DP variants.
  - To learn probabilities of those variant variables from data, it is necessary to annotate the data accordingly. If uninterested in variants, the intermediate variables could be omitted and all MPs involved wired directly to the DP variables.
- Probabilities are computed using Bayes theorem, where a hidden variable per variant can be calculated using knowledge of all observed variables [41].



- Unfortunately, it is questionable if new variant detection can be done efficiently via a knowledge-based system.
  - This is due to the fact that system is biased by the expert towards DP implementations known to him.
  - However, as the proposed system is more flexible than a classical rule-based approach due to the usage of MPs and probabilistic reasoning, it should be able to better detect new variants that share MPs with known variants.

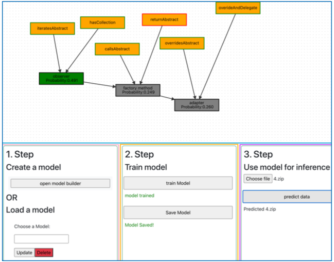
Metamodel Bayesian Network

- The output of both the ML and MP DPD subsystems is integrated into the Bayesian network HyDPD-B as shown in the example in the figure.
- To enable this, the result of the ML subsystem has to be interpreted as an observed variable in a network.
- Unfortunately, the system only allows binary variables, while the output cardinality of the ML system is dependent on the number of considered DPs.
- To avoid this, one can formulate variables in the following way:
  - a binary ML variable is associated with a model, as well as a specific DP.
  - If the prediction of the model equals the specified DP, the variable evaluates to true.



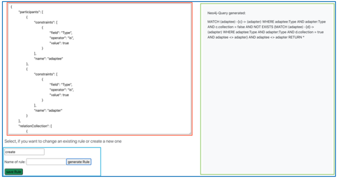
Realization Aspects

- Software used to realize the solution included:
  - sklearn, numpy, pandas, matplotlib, seaborn, NetworkX, Pomegranate for the Bayesian network, Flask, Jupyter notebooks, Docker, Docker Compose, Neo4j, MongDB, ReactJS, and JointJS.
  - The core of the backend was realized in Python as a library, which contains all modules necessary to create the Bayesian networks and ML models for DPD.



Web-based User Interface (UI)

- Single Page Application (SPA).
  - Jupyter Notebooks can suffice as a frontend for research purposes, but could be inconvenient for SW developers, who would have to code in Python and know the API of the library.
  - Instead, our UI provides the ability to create Bayesian networks graphically and train them via graphical UI elements (top figure).
  - Step 1: create or load network and visualize the decision-making process
    Step 2: training the model
    Step 3: the data can be loaded and a prediction run.
- UI for DPRL rule editor shows JSON and CQL (bottom figure)



Micro Pattern (MP) Catalog (MPC) Realization

1) Override Abstract: Derived from the Adapter Cypher query, it is a general MP describing a method that overrides an abstract method.

```
MATCH (adapter:Type)-[:INHERITS*1]->(target:Abstract:Type),
(adapter)-[:HAS]->(adapter_op:Operation),
(target)-[:HAS]->(target_op:Operation),
(adapter_op)-[:OVERRIDES]->(target_op) RETURN *
```

2) Iterate: This MP simply queries if a participant iterates over another participant, and commonly occurs in the Observer DP.

```
MATCH (a)-[:ITERATES]-(b) RETURN *
```

3) Abstract Function Call: This MP describes a call of an abstract function. Such calls occur in the Observer DP, more precisely when a notify function calls an update function.

```
MATCH (c_notify)-[:CALLS]->(update:Operation:Abstract) RETURN *
```

4) Has Collection: This MP queries if there is a participant that owns a collection of abstract types. This MP is frequent in the Observer DP.

```
MATCH
(c_subject:Type)-[:HAS {collection: true}]->(observer:Type:Abstract)
RETURN *
```

Micro Pattern (MP) Catalog (MPC) Realization

5) Override & Delegate: This MP describes a function overriding a function and calling another function, and was extracted from the Adapter DP.

```
MATCH (adapter_op)-[:OVERRIDES]->(target_op),
(adaptee)-[:HAS]->(adaptee_op:Operation),
(adapter_op)-[:CALLS]->(adaptee_op)
WHERE adaptee_op <> target_op RETURN *
```

6) Double Inheritance: This MP describes double inheritance, used in Adapter DP instances. If the Adapter pattern is implemented in the static, class-based way, the Adapter participant should in some way inherit from the adaptee as well as from the target.

```
MATCH (c)<-[:INHERITS]-(a)-[:INHERITS]->(b) RETURN *
```

7) Overriding Method Creates: This MP describes a method that overrides another method and creates an object. It was extracted from the Factory Method DP.

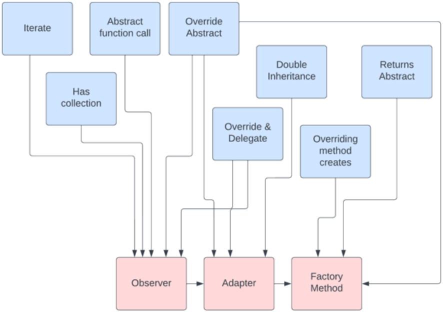
```
MATCH (creator_method)<-[:OVERRIDES]-(method)-[:CREATES]->(object) RETURN *
```

8) Returns Abstract: This MP matches methods that return an abstract class, and was extracted from the Factory Method DP.

```
MATCH
(concrete_create_op)-[:RETURNS]->(abstractproduct)<-[:INHERITS]-(concreteproduct)
RETURN *
```

MP Bayesian Network Realization

- Each DP is connected to relevant MPs.
- In HyDPD-GA, DPs were distinguished in a query by excluding certain features that would implicate another DP, as certain patterns exhibit a high degree of overlap in structure and behavior.
- Unfortunately, such exclusions make DPD more complex.
- To resolve this, output variables of frequently confused DPs are interconnected with each other.
- An example of the resulting network can be seen in the figure.



Bayesian network architecture for 3 DPs utilizing 8 MPs

Metamodel Bayesian Network Realization

- Leaf variables:
  - Corresponds to the result of a MP match against the graph. Thus, the value of a leaf variable can be calculated deterministically at inference time. The variable requires a binary output (False or True). While it is feasible to use continuous variables, it would make the system less comprehensible and interpretable.
- Hidden variables:
  - Cannot be directly detected like measurable variables. The output of a hidden variable depends solely on the input of parent variables. To allow a model to learn values of hidden variables from data, the data must annotated accordingly. A hidden variable can be expressed as a conditional table, which maps each combination of parent variables to a probability value (e.g., T/T->0.8, T/F->0.5, F/F->0.2). In practice, such annotations might indicate the specific pattern variants or participants involved in the pattern.
  - For DPD, hidden variables may correspond to following entities: DP probability that code is instance of a specific DP; DP variant probability that code is instance of a specific pattern variant; DP participant probability that code contains a DP participant; and MP pattern probability that code contains a specific MP.
- Query variables
  - We are not necessarily interested in all available hidden variables.
  - For DPD, we are specifically interested in the probabilities given to DPs. Consequently, in most use cases, query variables correspond to DPs.

Evaluation of HyDPD-B

- Used the same dataset as used for HyDPD [8]
- Due to resource constraints, we focused on three common patterns from each of the major pattern categories:
  - from the creational patterns, Factory Method;
  - from the structural category, Adapter;
  - from the behavioral patterns, Observer.
- 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.). They were manually verified and labeled as examples of a specific pattern.
- srcML supports these two popular programming languages and the mix of languages demonstrates programming language independence.
- For HyDPD-ML training data, we applied hold-out validation, selecting 60 of 75 projects (20 per pattern category), with between 60-75% of the code projects being in Java and the remainder in C#.
- To create the ML test dataset, the remaining 15 projects (5 per pattern, 3 in Java and 2 in C#) were duplicated and their signal words removed or renamed, resulting in 30 test projects (10 per pattern).

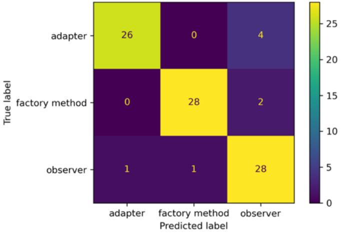
Evaluation of HyDPD-MP (Bayesian Network without ML)

Performance:

- Repeated cross-validation was used to test the performance of the rule-based system. Simple cross-validation showed high variance leading to inaccurate results. Thus, 5-fold cross-validation with 5 repetitions was used, resulting in 25 runs and a more accurate estimation.
- The mean was 0.917 and the median 0.944, with the distribution skewed due to outliers.
- Hence, accuracy of HyDPD-MP for these 3 DPs using an 8 MPs ruleset is on par with the 0.91 accuracy of our previous HyDPD-GA system [8].

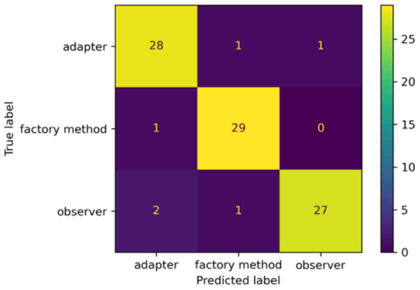
Confusion matrix:

- To determine if the results vary across different DPs, a confusion matrix was created using 5-fold cross-validation as shown in the figure.
- Adapter performed worse than the other patterns and was more frequently misclassified as Observer, an indication of some similarity between the DPs. Apparently, the ruleset does not properly distinguish Adapter from Observer. This result could likely be improved via better fitting Adapter rules, or via more restrictive Observer rules.



Evaluation of HyDPD-ML

- To evaluate HyDPD-ML, which utilizes graph embeddings, cross-validation was used, with the confusion matrix shown.
- Classification errors exist across all classes, yet no clear bias can be detected. Observer had the worst recall rate with 0.90, Adapter 0.93, and Factory Method with 0.97.



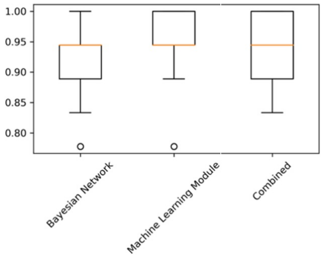
Evaluation of HyDPD-ML Variant detection

- DP variant datasets are difficult to acquire since most example DP projects intend to exemplify the reference DP. To evaluate HyDPD-ML for unknown pattern variant detection, DP variations were removed from the training dataset and moved to a test set containing only variations.
- As seen in the table, 6 out of 8 variations were correctly classified. The recall rate for Adapter was 1.0, Observer was 0.66, and Factory Method was 0.5.
- On average, accuracy is 0.75. While worse than the estimated general accuracy of 0.95, it shows HyDPD-ML is somewhat capable of classifying unknown pattern variations.

DP Variant	Predicted DP
Adapter 2	Adapter
Adapter 4	Adapter
Adapter 7	Adapter
Factory Method 17cs	Adapter
Factory Method 2	Factory Method
Observer 12	Observer
Observer 13cs	Factory Method
Observer 18cs	Observer

Evaluation of Combined HyDPD-B

- To evaluate the performance of the combined HyDPD-B, repeated cross-validation was performed. HyDBD-ML was trained on the same dataset as the Bayesian network.
- HyDBD-B (HyDPD-MP and HyDPD-ML combined) reached an accuracy of 0.944.
- While the Bayesian network is quite performant, it outperforms HyDPD-GA only by a very small margin.
- HyDPD-ML performs better than the Bayesian network.
- The rule set could be improved, as there is lot of potential gain by introducing more fitting rules.
  - This was not performed in the context of our current work as this could lead to a risk of manual overfitting of the available dataset.
- Combining the Bayesian network with the ML leads to a performance almost on the same level as ML itself.
- However, the new solution HyDPD-B is now more flexible for incorporating expert knowledge to continually improve and refine results



Conclusion

- This paper described our hybrid DPD solution concept HyDPD-B, which uses a Bayesian network to integrate a graph-based expert rule system using micropattern detection (HyDPD-MP) with a ML system (HyDPD-ML) using graph embeddings.
- Via a Bayesian network, inexact DP matching via probabilistic reasoning is supported with a finer rule definition granularity via micropatterns.
- The Bayesian network provides a flexible framework for probabilistic reasoning that is comprehensible and interpretable for humans.
- Our simple DP rule language (DPRL) was introduced to integrate developers as experts in defining DP and MP rules.
- Whereas HyDPD-MP can support DP localization and known variant detection via MPs, HyDPD-ML only indicates a DP is contained somewhere in the dataset.
- HyDPD-ML can detect *unknown DP variants*, yet with less accuracy than standard DPs.
- This could be improved with larger DP training and variant test datasets, but these remain challenging to acquire. Since the Bayesian system is dependent on manual knowledge engineering, future work will investigate its viability and scalability regarding DP variant detection.
- Future work includes expansion across all GoF DPs, measurements against benchmark pattern repositories and open source projects, and a comprehensive empirical industrial case study

References

1. 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.
2. 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.
3. E. Gamma, Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education India, 1995.
4. 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.
5. 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.
6. H. Yarahmadi and S. M. H. Hasheminejad, "Design pattern detection approaches: A systematic review of the literature," Artificial Intelligence Review, 53, pp. 5789-5846, 2020.
7. 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), pp. 27-32, IARIA XPS Press, 2020.
8. R. Oberhauser, "A Hybrid Graph Analysis and Machine Learning Approach Towards Automatic Software Design Pattern Recognition Across Multiple Programming Languages, " International Journal on Advances in Software, vol. 15, no. 1 & 2, year 2022, pp. 28-42. ISSN: 1942-2628.
9. 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.
10. B. Mayvan and A. Rasoolzadegan, "Design pattern detection based on the graph theory," Knowledge-Based Systems, vol. 120, pp. 211-225, 2017.
11. 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.
12. 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.
13. 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.
14. P. Pradhan, A. K. Dwivedi, and S. K. Rath, "Detection of design pattern using graph isomorphism and normalized cross correlation," Eighth International Conf. on Contemporary Computing (IC3 2015), pp. 208-213, IEEE, 2015.
15. 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.
16. 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.
17. 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.
18. R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, "Design pattern mining enhanced by machine learning," 21st IEEE In"l Conf. on Softw. Maintenance (ICS"05), IEEE, pp. 295-304, 2005.
19. 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.
20. 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.
21. 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.
22. 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.
23. 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.
24. 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.
25. M. Lebon and V. Tzerpos, "Fine-grained design pattern detection," IEEE 36th Annual Computer Software and Applications Conference, IEEE, pp. 267-272, 2012.
26. 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.
27. 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.
28. 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
29. 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.
30. 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.
31. K. Tu, J. Li, D. Towsley, D. Braines, and L. D. Turner, "Gl2vec: Learning feature representation using graphlets for directed networks," in Proceedings of the 2019 IEEE/ACM international conference on advances in social networks analysis and mining, 2019, pp. 216–221.
32. S. Sang et al., "Gredel: A knowledge graph embedding based method for drug discovery from biomedical literatures," IEEE Access, vol. 7, pp. 8404–8415, 2018.
33. J. Singh, S. R. Chowdhuri, G. Bethany, and M. Gupta, "Detecting design patterns: a hybrid approach based on graph matching and static analysis," Information Technology and Management, 23(3), pp. 139-150, 2022.
34. R. Barbudo, A. Ramírez, F. Servant, and J. R. Romero, "GEML: A grammar-based evolutionary machine learning approach for design-pattern detection," Journal of Systems and Software, 175, p. 110919, 2021.
35. M. Kouli and A. Rasoolzadegan, "A Feature-Based Method for Detecting Design Patterns in Source Code," Symmetry, 14(7), p. 1491, 2022.
36. 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.
37. N. Francis et al., "Cypher: An evolving query language for property graphs," Proc. 2018 International Conference on Management of Data, pp. 1433-1445, 2018.
38. S. Khwaja and M. Alshayeb, "Survey on software design-pattern specification languages," ACM Computing Surveys (CSUR), vol. 49, no. 1, pp. 1–35, 2016.
39. J. Smith and D. Stotts, "An elemental design pattern catalog," Technical Report TR-02–040, 2002.
40. A. Dempster, F. Petitjean, and G. I. Webb, "Rocket: Exceptionally fast and accurate time series classification using random convolutional kernels," Data Mining and Knowledge Discovery, vol. 34, no. 5, pp. 1454–1495, 2020.
41. K. P. Murphy, Machine Learning: A Probabilistic Perspective. MIT press, 2012.