Aalen University

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

Roy Oberhauser
Aalen University, Germany

IARIA ICSEA 2020

Software design patterns

- Application of known solutions to recurring software design problems
- Well-documented & popularized in the software development community, e.g. via
  - *Design patterns: elements of reusable object-oriented software* by E. Gamma et al. (a.k.a. GoF)
  - *Pattern-oriented software architecture* series by Wiley (a.k.a. POSA)
  - Portland Pattern Repository's Wiki
- Have brought about valuable improvements to and discussions around software design

Problems related to SW design patterns

- Design patterns are mostly described informally, no standardized terminology, naming, notation
- Implementations can vary widely and may not be obvious
  - Depending on the programming language, natural language of programmer, tribal community
  - Pattern structure and terminology awareness of the programmer, her/his experience, and their interpretation.
- Detection and documentation of these software design solution patterns has relied on experts
  - Experience, recollection, and manual analysis by experts.
- Some popular pattern books were published over 25 years ago
  - Many million lines of code have since been programmed, much of it not open source or accessible
- The code has not been subjected to any comprehensive analysis.
- Project documentation about applied patterns, if existent, may be inconsistent with the current source code reality and thus not necessarily dependable
  - E.g., prescriptive documentation of intentions, adaptations during development, maintenance changes
  - Known pattern variants may occur, patterns may evolve over time with technology, and in fact new patterns may unknowingly be developed that the experts may be unaware of.
- The investment for manual pattern extraction, recovery, and archeology is not economically viable

Motivation for research

- Automated feature extraction of software design patterns from documentation or code repositories is not yet commonly available among popular SDLC tools
- Insight into actual pattern usage could be beneficial
  - Meta-level: identifying which patterns are used how frequently and determine pattern trends and evolution
  - Application-level: avoiding unintended pattern degradation/erosion and associated technical debt, quality, and maintenance issues
- Research has attempted to find automated techniques that work
  - Most of the published techniques have not applied machine learning (ML) to this problem area
  - One implicit challenge for most approaches is to demonstrate just basic coverage of all of the GoF patterns - which very few, if any, achieve

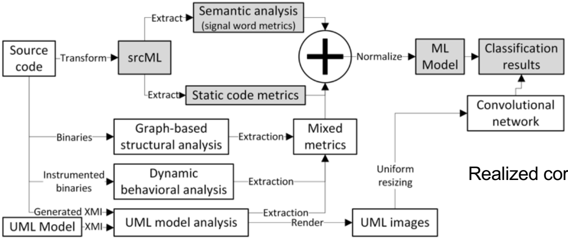
DPDML Solution

Design Pattern Detection using Machine Learning (DPDML):

- A generalized and programming language independent approach
- Automated design pattern detection based on ML

DPDML solution approach

Hypothesis: utilizing all available data, especially design pattern-related metrics, and feeding this input into an artificial neural network (ANN) or other ML models, we can achieve suitable classification accuracy for automated detection



Realized core DPDML-C (shown in grey)

DPDML Principle: ML model

- 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.
  - The realization may apply or combine any ML model that suites the situation, be it AutoML, unsupervised, supervised learning, etc.
  - In our current realization, 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.

DPDML Principle: Graph-based analysis

- ML model
- Graph-based analysis (GBA)
  - Could query aspects, enhance classification results, and support manual pattern verification
  - Code repositories are analyzed using graph-based tools like jQAssistant and various metrics extracted.

DPDML Principle: Programming language-independent

- ML model
- Graph-based analysis (GBA)
- Programming language-independent
  - Source code is converted into an abstracted common format for further processing.
  - We can then, e.g., extract various metrics in a common fashion, independent of the original programming language syntax.
  - Our realization utilizes srcML, thus our realization can currently support any programming languages that map to the srcML XML-based format, including C, C++, Java, and C#.

DPDML Principle: Semantic analysis

- ML model
- Graph-based analysis (GBA)
- Programming language-independent
- Semantic analysis
  - Common pattern signal words from the source code can be used as an indicator or hint for specific pattern usage.
  - Our realization utilized the signal words in the table, and supports German, Russian, and French.

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

DPDML Principle: Static code metric extraction

- ML model
- Graph-based analysis (GBA)
- Programming language-independent
- Semantic analysis
- Static code metric extraction
  - Various static code metrics are utilized to detect and differentiate design patterns
  - Our realization utilizes those shown in the table

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

Inspired by Uchiyama et al.

DPDML Principle: Dynamic analysis

- ML model
- Graph-based analysis (GBA)
- Programming language-independent
- Semantic analysis
- Static code metric extraction
- Dynamic analysis
  - Tracing runtime code behavior can detect behavioral similarities in event sequencing, especially for the creational or behavior patterns
  - Event and related runtime metrics can be extracted

DPDML Principle: UML structural analysis

- ML model
- Graph-based analysis (GBA)
- Programming language-independent
- Semantic analysis
- Static code metric extraction
- Dynamic analysis
- UML structural analysis
  - Extract indicators/signal words/metrics from XMI structures
  - Generate UML from code - but code has the basis already
  - A convolutional network could analyze UML images for similarities to support pattern classification

DPDML Principle: Metric normalization

- ML model
- Graph-based analysis (GBA)
- Programming language-independent
- Semantic analysis
- Static code metric extraction
- Dynamic analysis
- UML structural analysis
- Metric normalization
  - Metric value ranges normalized to scale 0-1 scale

DPDML Principles

- ML model
- Graph-based analysis (GBA)
- Programming language-independent
- Semantic analysis
- Static code metric extraction
- UML structural analysis
- Dynamic analysis
- Metric normalization

Realization challenges: Comprehensive DPDML

- Due to unexpected obstacles and project resource constraints, only a partial realization of the comprehensive DPDML was achieved as explained

Realization challenges: Comprehensive DPDML

- UML structural analysis
  - Almost none of the 60 repos used for the evaluation provided UML diagrams
    - Even they had, manual code-to-UML validation would be time-consuming. Signal words may exist in one and not in the
  - In our opinion, larger commercial closed-source projects are more likely to include UML documentation
  - Since little benefit could be had for our evaluation, it was not yet realized and will be addressed in future work

Realization challenges: Comprehensive DPDML

- Dynamic analysis
  - Differing runtime environments, languages, libraries, concurrent processing
  - Requires runnable binaries, which not every evaluation repo had
  - Requires specialized tooling to acquire behavior tracing data
    - No standard formats or tools exist in this area
  - Compute-intensive and time-consuming to manually setup and acquire pattern-related traces
  - Ensuring the patterns are actually substantially executed
    - Can be an issue for larger projects.
  - Gathering sufficiently large training sets for ML
  - An interesting academic exercise to improve our understanding
    - Yet probably impractical and not economically viable for practitioners
  - Not yet realized and will be addressed in future work

Realization challenges: Comprehensive DPDML

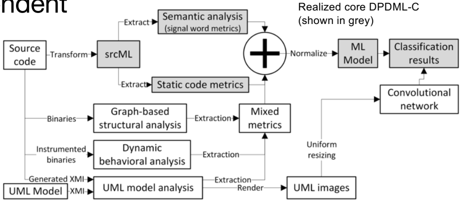
- Graph-based analysis (GBA)
  - GBA tools typically require compiled binaries for analysis
    - Not all of our evaluation repos consisted of compiled/compilable code
  - GBA tools typically
    - Programming language-specific, IDE-specific, and assume GUI-based human-interaction
    - Not geared for automated analysis of many projects in various languages
  - Reverse-engineering or code analysis tools
    - Often commercial
    - Missing a command-line mode
    - Limited use for automated analysis situations in our context
  - Will be addressed in future work

Realization challenges: Comprehensive DPDML

- Graph-based analysis (GBA)
  - GBA tools typically require compiled binaries for analysis
    - Not all of our evaluation repos consisted of compiled/compilable code
  - GBA tools typically
    - Programming language-specific, IDE-specific, and assume GUI-based human-interaction
    - Not geared for automated analysis of many projects in various languages
  - Reverse-engineering or code analysis tools
    - Often commercial
    - Missing a command-line mode
    - Limited use for automated analysis situations in our context
  - Will be addressed in future work

Realization: DPDML-C core

- Determine if the core of the DPDML solution and the following principles work as intended
  - ML model
    - Initially an ANN was used for our investigate
  - Programming language-independent
  - Semantic analysis
  - Static code metric extraction
  - Metric normalization

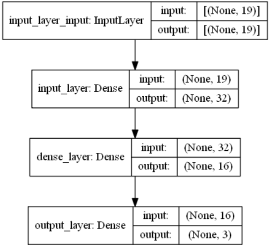


Realization: DPDML-C details

- Due to resource and time constraints, initially focused on learning to detect a single pattern from the categories:
  - Structural: Adapter
  - Creational: Factory
  - Behavioral: Observer
- Scope to be expanded in future work
- Python, TensorFlow, Keras
- Semantic analysis
  - Signal words (Python translate used): English, German, French, Russian

Realization: DPDML-C ANN

- Input layer size matches data points: 7 metrics and 12 semantic match values (19 total). The input model structure is a numpy array:
  - [NOC, NOF, NOSF, NOM, NOSM, NOI, NOAI, ASW1, ASW2, ASW3, ASW4, FSW1, FSW2, FSW3, FSW4, OSW1, OSW2, OSW3, OSW4]
  - First 7 values correspond to metrics table (right)
  - Rest indicate number of signal word matches (bottom table) SW=Signal Word, A=Adapter, F=Façade, and O=Observer, 1-4 implies table column
- Output layer: 3 neurons corresponding to the 3 design patterns
- Activation method: "Softmax"
- "Adam" with default values used as optimizer
- No regularization was applied in each layer
- Loss function: sparse categorical crossentropy
- ANN size should fit problem size
  - Small ANN structure adjustments showed no significant performance impact, whereas significantly increasing the neuron count or layer count negatively impacted results.
  - 2 hidden layers and 48 neurons: 1st layer has 640 parameters, the 2nd layer 528, and output layer 51, resulting in 1219 parameters that are adjusted during training



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

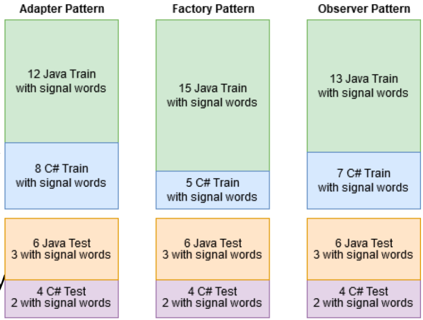
Pattern	Signal Words			
Adapter	Adapter	adaptee	target	adapt
Factory	Factory	create	implements	type
Observer	observer	state	update	notify

Realization: ANN training

- ANN trained in epochs
  - The complete training set is sent through the network whereby weights are 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
- Design pattern training sets considered:
  - Pattern-like Micro-Architecture Repository (P-MARt):
    - Includes a collection of microstructures found in different repositories such as JHotdraw and JUnit
    - Patterns are intertwined with each other, so they do not provide isolated example specimens for training the ANN
  - The Perceptrons Reuse Repositories: results were not available on website during our realization
- Reason for ANN:
  - DPDML intent initially much broader scope for data pattern mining
  - Expected a large supply of sample data.
  - Interested in determining if we could train an ANN to detect these patterns with relatively few samples
- Unexpected additional resource and time involved due to manually searching for pattern samples resulted in:
  - Reduction in number of design patterns trained and tested (see future work)
  - No comparison with alternative ML classification schemes (see future work)

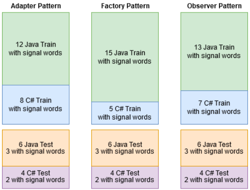
Evaluation: Dataset

- Used 75 small single-pattern code projects from public repositories (github, pattern book sites, MSDN, etc.)
  - 49 in Java
  - 26 in C#
  - Demonstrates the programming language independent principle
  - Inequality in ratio due to language popularity and age
- Evenly distributed into 25 unique code projects per pattern.
- They were specifically labeled as examples of these patterns, and manually verified.



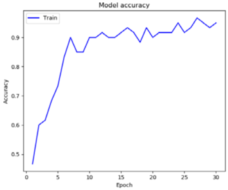
Evaluation: ANN training

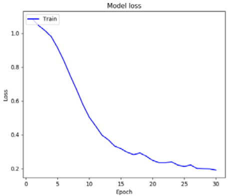
- *Supervised training dataset*: 20 projects per pattern category (60 of the 75 total)
  - 60-75% Java projects (green) and the remainder in C# (blue) as shown on right
- *Test dataset:* allocated the remaining 15 of the 75 projects (5 per pattern category (3 in Java (orange) and 2 in C# (magenta))
  - To evaluate signal word pattern matching impacts on ANN results:
    - Duplicated projects and removed/renamed signal words
      - Thus 6 Java (orange) and 4 C# (magenta) projects per pattern/category (bottom of figure)
  - Resulted in 10 test projects per pattern (30 total)



Evaluation: ANN accuracy and loss

- Accuracy improves from 47% to 95% in the first 7 training epochs (top right figure), thereafter fluctuating between 85-95% with a peak of 96.7% in the 27th epoch
- 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
  - Indicates a positive learning behavior and implies that with the given data points, it is finding structures and values that allow it to differentiate the three design patterns from each other
- Thus, we stopped the training at 30 epochs
  - Training took 2-45 seconds depending on the underlying hardware environment
- Training summary:
  - Considering that the worst case of random guessing would result in an accuracy of 33%, 97% accuracy is significantly better and shows the potential of the approach
  - 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 2 out of the 60 total code projects spread evenly across the three design patterns are incorrectly classified





Evaluation: Testing

- The test dataset used 15 unique code projects (5/pattern) duplicated and signal words removed/renamed, resulting in 30 code projects.
  - Signal words removal for determining degree of dependence of the ANN on signal words
- Accuracy dropped to 83.3%: 25 of 30 patterns correctly identified
- Loss increased to 0.4060: loss in confidence of categorization
- Deterioration expected when working with unfamiliar data
- Result: ANN able to use its learned knowledge from training to correctly classify a majority of unknown projects (25 out of 30)

Evaluation: Confusion matrix (for 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%			

- Precision column indicates how many of the predicted labels are correct
- 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 with the other patterns Possible explanation: metrics used may better differentiate more complex patterns
  - Other patterns had less precision (81% or 75%), but a better recall of 90%
  - The overall F_1 score is 0.83
- Signal word influence: hypothesis that signal words would improve results unfounded
  - Classification precision unaffected: 12 projects with and 13 without were 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
- 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

Conclusion

- DPDML provides a generalized and programming language-independent approach for automated design pattern detection based on ML
- Our realization of the DPDML-C core of the solution approach shows the feasibility of key aspects of DPDML: ML model, programming language-independent, semantic analysis, static code metric extraction, metric normalization
- Our realization of the core DPDML-C shows its feasibility for source code-based analysis
- Evaluation with 75 unique Java and C# code projects acoss 3 common GoF pattern categories
- Supervised training on 60 unique Java and C# code projects achieved an accuracy of 83% and loss of 0.4060 on testing 15 unfamiliar code projects (which duplicated with signal word modifications)
  - Investigated the feasibility and potential of ANN for automated design pattern detection
  - The accuracy result was achieved based only on static analysis, without involving cost-intensive behavioral analysis
- For the 3 patterns, signal words did not improve results, so other pattern characteristics can potentially suffice as indicators
- DPDML shows promise for extending the automated detection to other patterns

Future work

- Investigate the inclusion of additional pattern properties and key differentiators to improve the results even further, including:
  - Analyzing the network classification errors to optimize 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++
  - Extending 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
  - Evaluate pattern detection when patterns are intertwined
  - Evaluate accuracy, performance, and practicality on large projects
  - Investigate the detection of new design patterns and variants to the traditional patterns
  - Apply cross-validation and consider alternative classification schemes such as Naïve Bayes, Decision Tree, Logistic Regression, and SVMs
- An empirical industrial case study

Thank you!