Design Pattern Mining Enhanced by Machine Learning

Rudolf Ferenc, Árpád Beszédes, Lajos Fülöp and János Lele
University of Szeged, Department of Software Engineering
{ferenc|beszedes}@inf.u-szeged.hu
{Fulop.Lajos|Lele.Janos.1}@stud.u-szeged.hu

Abstract

Design patterns present good solutions to frequently occurring problems in object-oriented software design. Thus their correct application in a system's design may significantly improve its internal quality attributes such as reusability and maintainability. In software maintenance the existence of up-to-date documentation is crucial, so the discovery of as yet unknown design pattern instances can help improve the documentation. Hence a reliable design pattern recognition system is very desirable. However, simpler methods (based on pattern matching) may give imprecise results due to the vague nature of the patterns' structural description. In previous work we presented a pattern matching-based system using the Columbus framework with which we were able to find pattern instances from the source code by considering the patterns' structural descriptions only, and therefore we could not identify false hits and distinguish similar design patterns such as State and Strategy. In the present work we use machine learning to enhance pattern mining by filtering out as many false hits as possible. To do so we distinguish true and false pattern instances with the help of a learning database created by manually tagging a large C++ system.

Keywords

Columbus, C++, Design Patterns, Machine Learning, StarOffice

1 Introduction

The correct use of *Design Patterns* in software development is commonly treated as a premise for the high quality of the design in terms of, among other aspects, reusability and maintainability. Well-established and documented design patterns exist in various fields of software development. One of the most commonly recognized pattern catalogs was compiled by Gamma *et al.* [11], which describes patterns used in object-oriented analysis and design. We will deal with these patterns in this work. Any system with a high-quality design contains occurrences of design patterns, irrespective of whether they are introduced by the designer intentionally or unwittingly. Whatever the case, knowing about the instances of patterns in a software system may be of great help during software maintenance (for instance, to better understand the system's structure and workings). Unfortunately, in many cases the pattern usage is poorly documented, so automatic or semi-automatic procedures for discovering design patterns in undocumented software can be very helpful to maintainers.

Design pattern recognition from existing systems is used for various purposes, the main areas being *maintenance* (e. g. Wendorff determines maintenance problems by identifying incorrect and extreme usage of patterns [24]) and *program comprehension* (e. g. Campo *et al.* use design pattern recognition for framework comprehension [7]). It is important to concentrate on pattern instances during these activities since, as Biemann *et al.* found, participant classes in patterns change more rapidly than other classes in a system [5].

The methods for discovering design pattern instances from existing source code are also varied. One of the first methods was an inductive one published by Shull *et al.* [18] that relies heavily on manual checking. Other authors use rules to discover design patterns from source code. In this case recognition rules are built from the design patterns' structural features (Niere *et al.* [15]). Pattern matching on graph representations of the source code using the structural features of design patterns represents another significant class of methods. We also approached the problem this way in [4]. The precision of pattern discovery is improved by some researchers using dynamic analysis which, apart from the structural information, also considers runtime information about the system under investigation [22, 23]. However, all these methods use a fixed pattern library containing the descriptions of the patterns. In contrast to this, Tonella and Antoniol proposed a different approach for recognizing design patterns in [21]. They search the recur-

Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)
1063-6773/05 $20.00 © 2005 **IEEE**

rent patterns in the source code automatically, after which they compare the patterns found with the known design patterns. This way they were able to find even those patterns that were not part of any standard pattern catalog.

The problem with the more common approaches to pattern recognition (based on pattern matching) is that they are inherently too permissive in a sense that they produce many false results in which some code fragments are identified as pattern instances that share only the *structure* of the pattern description. This is due to the fact that the patterns themselves are given using conventional object-oriented concepts, such class diagrams containing abstract classes and methods, generalization, polymorphism, decoupling concrete responsibilities through references to abstract classes, and so on. This leads to structures that are in many ways quite similar to each other (consider the structures of, say, Bridge vs. Adapter Object,[1] Composite vs. Decorator or State vs. Strategy). Furthermore, such common structures may appear even for code fragments that were not designed with the intent of representing any specific design pattern, but make use of the above mentioned techniques simply as good object-oriented solutions to some other problems. The distinction between such true and false pattern instances and between different patterns with the same structures can be made only by applying more sophisticated methods that involve a deeper investigation of the implementation details and its environment, i. e. to guess its real purpose. This naturally leads us to apply machine learning methods to refine the recognition capabilities of a conventional pattern matching-based method.

In previous work [4] we presented a method for mining design pattern instances using the Columbus reverse engineering system [9]. Columbus represents a C++ source code in the form of an Abstract Semantic Graph (ASG) [8], and our approach to locating patterns was to try to find the patterns' structures in the graph using a custom-made pattern matching method. In this system the structure of a design pattern is stored using our special format based on XML. This format, the Design Pattern Markup Language (DPML), is appropriate for storing the structure of a design pattern by capturing the prescribed design elements found in the conceptual description. This includes classes, their required attributes and operations, and the relations between the participants. Finding a design pattern via this approach means, then, applying our graph matching algorithm with the DPML description to the ASG. Clearly, since other information besides the mere structure of a graph fragment is not used, many false pattern instances could be found using this approach.

[1]The Adapter design pattern has two variants: a *class* version and an *object* version that differ in the way the adaptation is achieved: using multiple inheritance or object composition. We investigated the latter one and in the following we refer to it as the *Adapter Object* pattern.

In the present article we will overview the enhancement to our pattern matching-based approach. We used machine learning methods to further refine the pattern mining by marking the pattern instance candidates returned by the matching algorithm either as true or false instances, and this way produced better results. Note that with this approach we are not able to find new instances designed as pattern instances but their structure differed to some extent from the "textbook" structures. This kind of enhancement is a completely different research topic. In other words, we further *filter* the pattern instances returned by the basic algorithm by eliminating false hits.

Our approach in a nutshell is to analyze the candidates returned by the matching algorithm, taking into account various aspects of the candidate code fragment and its neighborhood, such as whether a participant class has a base or not, or how many new methods a participant class defines besides a participating method. The information corresponding to these aspects is referred to as *predictors*, whose values can be used in a machine learning system for decision making. We employ a conventional learning approach, that is we first manually tag the candidates as true or false instances and calculate the values of the predictors on the candidates. Then we provide these to some learning system (we conducted our experiments using two methods, a decision tree-based one and a neural network approach). This in turn provides a model that incorporates the acquired knowledge, which can later be used for pattern mining in unknown systems.

We performed our experiments on *StarWriter* [20] as the subject system for pattern mining and we searched for the *Adapter Object* and the *Strategy* design patterns. The choice for these two patterns was made because there were enough candidate instances of them found by the matching algorithm in the source code and because these candidates were sufficiently variegated.

Our results are promising, and they suggest that machine learning can be successfully applied to the design pattern mining problem. We achieved learning precisions of 67–95% and with the model obtained 51 of the 59 false hits could be filtered out of a total of 84 hits of the Adapter Object pattern, say.

The paper is organized as follows. In the next section we will overview some other works having similar objectives to ours. In Section 3 we will look at the design patterns investigated, namely Adapter Object and Strategy. Section 4 describes our approach, while Section 5 is devoted to the two machine learning approaches we employed. In Section 6 we will present the main results of our work, then in Section 7 we offer conclusions and suggestions for future study.

2 Related Work

Many researchers have sought to develop methods for recognizing design pattern instances from a high-level design (e. g. UML diagrams) or the already existing source code. In this section we will briefly review some papers that have similar goals to ours.

Guéhéneuc et al. [12] introduced a method for reducing the search space for design patterns. First, they analyzed several programs manually and searched for source classes that act together as design patterns and set up a repository from them. Afterwards they parsed these programs with tools to obtain models of their source code, and computed metrics from these models (like size, cohesion and coupling). In the next step they ran their rule learner algorithm that returned a set of rules characterizing the design pattern participants with the metric values. This way they obtained rule sets (called a fingerprint) for the participant classes of the design patterns. Based on these fingerprints unknown classes could be characterized. They then integrated this fingerprint technique with their constraint-based tool suite to reduce the search space. Their work is in some sense similar to ours, but we made use of machine learning after the structural matching phase (taking into consideration the whole design pattern and not just individual classes) to filter out false instances.

In [10], design pattern detection was accomplished via the integration of two existing tools – Columbus [9] and Maisa [16]. The method combined the extraction capabilities of the Columbus reverse engineering system with the pattern mining ability of Maisa. First, the C++ code was analyzed by Columbus. Then the facts collected were exported to a clause-based design notation understandable to Maisa. Afterwards, this file was analyzed by Maisa, and instances were searched for that matched the previously given design pattern descriptions. Maisa approached the recognition problem as a constraint satisfaction problem.

Tonella and Antoniol [21] presented an interesting approach for recognizing design patterns. They did not use a library of design patterns as others did but, instead, discovered recurrent patterns directly from the source code. They employed concept analysis [19] to recognize groups of classes sharing common relations. The reason for adapting this approach was that a design pattern could be considered as a formal concept. They used inductive context construction which then helped them find the best concept.

Antoniol *et al.* introduced pattern recognition by using metrics [2]. They analyzed source code and class diagrams and created AOL (Abstract Object Language) specifications containing information about classes, their members and relations. The design pattern descriptions were also stored in AOL format. Next, they created an AST (Abstract Syntax Tree) from the AOL specification. Afterwards, they computed metrics for each candidate class from the AST and created a set for each participant class in the searched design pattern containing only those candidate classes which met the participant classes' metric conditions (these conditions were set up manually). They significantly reduced the search space this way. Finally, they checked the required structural relations among the candidate classes in these sets. They could also verify the consistency between the code and the design. They tested their system on public and industrial systems with good results.

Albin-Amiot *et al.* [1] introduced a Pattern Description Language (PDL) that was suitable for detecting design patterns from source code and also for generating source code. Their system was also able to detect some distorted versions of design patterns and could repair them automatically with the aid of a source-to-source transformation engine.

Keller *et al.* [14] argued that design patterns are the bases of many of the key elements of large-scale software systems, so to comprehend these systems they needed to recover and understand design patterns. They emphasized not only the design's structure but its rationale too. They utilized the SPOOL environment which provided tools for analyzing existing source code and recovering design components like design patterns. They implemented query mechanisms that could recognize the structural descriptions of patterns in the source code models. The SPOOL environment gave some visual information about the query's results (the found design patterns) and information about the design pattern's class diagram to discover the pattern's structure and documentation about its intent and motivation. They used this environment with three industrial systems and searched for three design patterns (Template Method, Factory Method and Bridge). They checked the intent of the design patterns found and noticed that the discovered design patterns' intents did not necessarily correspond to the original design patterns' intents.

Asencio *et al.* [3] used the Imagix [13] tool to parse the source code. The tool built a database of program entities and relationships. They introduced a recognizer specification language where they made declarative specifications – logical conditions – to describe design pattern structures. Afterwards, their tool called Osprey automatically generated Python source code from these declarative specifications, which searched patterns in the database generated by Imagix. They tested their system on several software systems and obtained promising results but also found false instances. They classified the causes of these false hits. The first class of problems were the front end analyzer errors. The second class was the pattern ambiguity. The cause for these false hits were the structural similarities among design patterns, like those of the Decorator and Proxy patterns. The third and last class were the partial patterns. There were many situations where the full pattern was not present in the code under analysis.

Wendehals [22] improved design pattern recognition with dynamic analysis. First, the author performed static analysis and labelled each pattern candidate with a fuzzy value that represented the correctness of the candidate. Afterwards, false candidates were ruled out by dynamic analysis and by using these fuzzy values. Wendehals also employed this technique in [23] where it was demonstrated how to distinguish between patterns having the same structure (like the State and Strategy patterns) by dynamic analysis.

Campo *et al.* [7] utilized design pattern recognition for framework comprehension. They concluded that some design patterns could be distinguished only by their dynamic behavior, because their structures were the same (e. g. Composite vs. Decorator and State vs. Strategy).

3 Studied Design Patterns

We performed our experiments on StarWriter (containing more than 6,000 classes), the text editor of the StarOffice suite [20]. We applied our design pattern mining approach presented in [4], but each experiment was repeated since both the C++ front end and the pattern mining algorithms have improved a lot since then. Using the pattern-matching algorithm of Columbus we first found several hundred pattern instances – which we treat in the present work as candidates because they will be further filtered with machine learning methods to provide more accurate results.

We chose two design patterns out of 16 patterns handled by the matching algorithm for the experiments. The final decision fell on the structural pattern *Adapter Object* and the behavioral *Strategy* pattern since these two occurred most frequently in the results. This choice was appropriate too because, after the manual investigation of the candidates, we found that there were enough positive and negative examples. Moreover, the candidates and their contexts were sufficiently different to train the machine learning systems successfully. These two patterns are good examples of how general the structural descriptions of patterns can be in terms of general object-oriented features, and how much useful the deeper information can be for their recognition.

3.1 Adapter Object

The aim of the Adapter pattern is to "convert the interface of a class into another interface that clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces" [11].

The Adapter pattern has four participants (see Figure 1). First, the Target class defines the domain-specific interface that the Client uses. Client in turn represents the class collaborating with objects that conform to the Target interface. Next, the Adaptee class describes an existing interface that needs adapting, and finally Adapter is the class that adapts the interface of Adaptee to the Target interface.

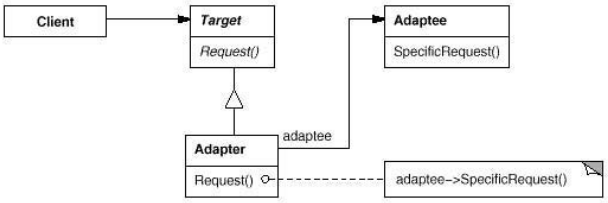

Figure 1. The Adapter Object design pattern

There are two forms of the Adapter pattern, namely Class and Object. The former uses multiple inheritance to adapt one interface to another, while the latter uses composition for the same purpose. We employ Adapter Object in our experiments.

It may be seen that this structure, the delegation of a request through object composition, is a quite common arrangement used by object-oriented systems and so only a more detailed analysis may spot the real instances of this pattern.

3.2 Strategy

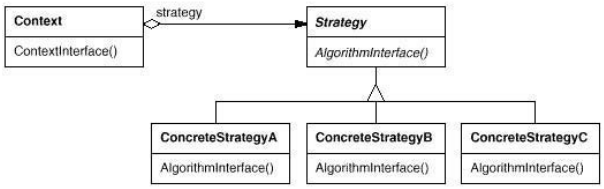

Figure 2. The Strategy design pattern

The intent of the Strategy pattern is to "define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it" [11].

The Strategy pattern has three participants (see Figure 2). The Strategy class declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy, which implements the algorithm using the Strategy interface. The Context class is configured with a ConcreteStrategy object and maintains a reference to a Strategy object.

The fact that the implementation of the algorithm interface is done simply by realizing the Strategy interface with inheritance and method overriding suggests that this pattern also requires a more detailed analysis to distinguish its true instances from the false ones.

4 Our Approach

In this section we will describe how we employed two machine learning methods to refine our results given in [4]. First, we will briefly present the Columbus framework. Next, we will describe the predictors we formulated for the machine learning, then we will overview the learning process used.

4.1 Columbus

Columbus [9] is a reverse engineering framework that, among other things, is able to recognize design patterns from C++ source code. The design patterns to be searched for are described in DPML (Design Pattern Markup Language) files which store information about the structures of the design patterns.

Columbus recognizes design patterns in the following way. First, it analyzes the source code and builds an Abstract Semantic Graph (ASG) that contains all the information about the source code. Then Columbus loads a DPML file which also basically describes a graph. Afterwards it tries to match this graph to the ASG using our special algorithm described in [4].

Columbus finds every design pattern instance that has the structure corresponding to the DPML file but it does not consider the found design pattern's intents. Actually, when Columbus finds a Strategy pattern it also finds a State pattern because the structures of these two patterns are identical, hence their DPML descriptions are also identical. It may also happen that Columbus identifies classes which are accidentally related in the same way as in a design pattern.

Now we will introduce a method that is able to refine the basic algorithm by removing false hits from the discovered design pattern instances by also considering their intended purpose.

4.2 The Predictors

Every design pattern has features that do not belong to its structural description. We retrieve this kind of information from the source code and give them to a learning system. We call these collected values predictors. In the following we will present the predictors that we formulated for the two design patterns we studied. We also experimented with other predictors, but these proved to be the most effective.

Adapter Object Design Pattern Predictors

- *A1 – PubAdapteeCalls predictor* shows how many public methods of the Adapter candidate class contain a method call to an Adaptee candidate class. Since the main purpose of the Adapter pattern is to adapt the Adaptee class to the Adapter class, we reach the Adaptee objects through the Adapter ones. So most public methods of an Adapter candidate should contain a method call to an Adaptee candidate.
- *A2 – PubAdapteeNotCalls predictor* shows how many public methods of the Adapter candidate class do not contain a method call to an Adaptee candidate class. We assume that the A1 value is greater than A2, because the Adapter's intent is to adapt the functionality of Adaptee so the Adapter must have more public functions that call Adaptee than those that do not.
- *A3 – NotPubAdapteeNotCalls predictor* shows how many non-public methods of the Adapter candidate class do not contain calls to an Adaptee candidate. We assume that if A2 is greater than A1 then the private or protected methods are responsible for calling the Adaptee.
- *A4 – MissingAdapteeParameter predictor* shows how many constructors of the Adapter candidate class do not get an Adaptee candidate as a parameter. The Adapter is likely to get the Adaptee object that it will manage via its constructors, so it should be zero or a low value.
- *A5 – AdapteeParameterRatio predictor* shows the ratio of the constructors of the Adapter candidate class that get an Adaptee candidate object as a parameter. We assume it should be one, or close to one.
- *A6 – NewMethods predictor* shows how many new methods the Adapter candidate class defines. The Client uses the Adaptee through the interface of the Target class, so since the purpose of the Adapter class is to define the methods of Target, it needs not adding its own new methods.

Strategy Design Pattern Predictors

- *S1 – InheritanceContext predictor* shows the number of children of the Context candidate class. It should be a low value, because otherwise the pattern would be more similar to the Bridge pattern than to Strategy.
- *S2 – IsThereABase predictor* shows the number of the Strategy candidate's parents. We assume that Strategy does not have parents because it provides the interface to change the strategy.
- *S3 – Algorithm predictor* this predictor investigates the ConcreteStrategy candidate classes. It represents a value based on the ConcreteStrategy candidates' algorithmical features like the number of loops and recursions. We suppose the more algorithmical features it has, the higher the probability of it being a true ConcreteStrategy.
- *S4 – ConcreteStrategy predictor* shows the number of ConcreteStrategy candidates discovered. If this value is low the main advantage of the Strategy pattern is lost.

- *S5 – ContextParam predictor* shows the number of methods of the Context candidate which have a Strategy parameter. Usually the Context class forwards the client's requests to Strategy so that the client can select the Strategy object at runtime. Thus Context should have at least one method with a Strategy parameter.
- *S6 – InheritanceStrategy predictor*: shows the number of direct descendants[2] of the Strategy candidate class. This value should be close to S4, but in any case it must be smaller.

4.3 The Learning Process

Using the predictors defined we can produce the decision models with the machine learning systems on a manually tagged training set. These decision models can then be integrated into the current pattern mining system, to provide more precise outputs.

In the following we will overview the concrete steps in this learning process. It consists of four consecutive steps, which are the following:

1. *Predictor value calculation.* Columbus creates an ASG from the source code, and finds the design pattern instance candidates that conform to the actual DPML file which describes the structure of the searched pattern. In this first step we calculate our predictor values from the ASG and save them to a predictor table file. This file is basically a table containing the predictor values in a row for each candidate design pattern instance.
2. *Manual inspection.* Here we examine the source code manually to decide whether the design pattern instance candidates are true or false hits. Then we extend the predictor table file with a new column containing the results of the manual inspection.
3. *Machine learning.* First we process the predictor table file and classify the predictor values according to their magnitude to achieve better learning results.[3] Next, we perform the training of the machine learning systems (which will be described in the next section). The outputs of these systems are model files which contain the acquired knowledge.
4. *Integration.* Finally, we integrate the results of machine learning (the model files) into Columbus to be able to make smarter decisions by filtering the design pattern candidates. This way Columbus should report much fewer false design pattern hits to the user.

[2]Note that this value is not the same as that of S4, because the ConcreteStrategy-s may be located deeper in the class hiearchy, and not all children of Strategy are necessarily ConcreteStrategy-s.

[3]We divide the values into equal intervals and use the classes corresponding to these intervals as the input for the learning algorithms.

We applied the above-described steps to StarWriter with the two design patterns chosen, and performed some experiments and calculations regarding the efficiency of learning. These will be discussed in Section 6.

5 Used Machine Learning Approaches

We employed two machine learning systems for acquiring knowledge from the predictor sets discussed in the previous section. Both systems produce a model that will be used for decision making. Besides the same predictor sets these two algorithms are also used in the same way so we were able to test both in the same environment.

They represent some of the most popular approaches in the field of machine learning, one being a decision tree and the other a neural network. The system we used for the former was C4.5 (which uses an enhanced version of the ID3 algorithm), while for the latter it was the Backpropagation algorithm.

5.1 Decision Tree

C4.5 is an enhanced implementation of the ID3 algorithm that was proposed by Quinlan in 1993 [17]. The C4.5 algorithm makes use of a variant of the rule post-pruning method to find high precision hypotheses to the target concept of the learning problem. It generates a classification-decision tree for the given data set by recursively partitioning the data. Training examples are described by attributes (predictor values) whose choice for a given tree node depends on their information gain at each step during the growth of the tree.

Some of the features of the algorithm are that it results in smaller decision trees, it uses a depth-first strategy, and that over-fitting is allowed – meaning that the growth of the tree continues until it best fits the training data. After the tree has been built up it will be converted into an equivalent set of rules, all of which incorporate tests of the predictor values and that will provide the output decision.

Lastly, pruning is applied to the rules to reduce the size of the tree by rearranging and removing similar branches. C4.5 can handle both discrete valued predictors and continuous ones as well, and also training examples with missing predictor values.

5.2 Neural Network

The Backpropagation algorithm [6] works with neural networks that are the means for machine learning, whose reasoning concept was borrowed from the workings of the human brain.

This algorithm uses more layers of neurons; it gets the input patterns and gives them to the input layers. Then it

computes the output layer (the output decision) from the input layer and the hidden (inner) layers. In addition, an error value is also calculated from the difference between the output layer and the target output pattern (the learning data).

The error value is propagated backwards through the network, and the values of the connections between the layers are adjusted in such a way that the next time the output layer is computed the result will be closer to the target output pattern. This method is repeated until the output layer and target output pattern are almost equal or up to some iteration limit.

6 Results

In this section we will present the results of our experiments concerning the precision of the learning methods and their effect on the accuracy of design pattern recognition.

The basic pattern matching-based algorithm with Columbus found 84 instances of Adapter Object and 42 of the Strategy pattern in StarWriter. Next, we performed a manual inspection of the source code corresponding to the found instances and provided a two-fold classification for each candidate instance; either as a true or false hit. Table 1 lists the statistics about this classification.

Pattern	Total hits	False hits	True hits
Adapter Object	84	59	25
Strategy	42	35	7

Table 1. Pattern instance candidates

This manually tagged list of instances was then used as the training set to the learning systems together with the calculated predictor values for each instance candidate, as described in Section 4.

In the following we will first overview our experiences with the investigation of the candidates and the relation to the actual predictor values, and then we will present our results about the learning efficiency.

6.1 Adapter Object Candidates Investigation

During the investigation of the Adapter Object candidates we found that they can be divided into groups that share some common features. We will show two examples of these groups.

Candidates belonging to the first group all have an Adapter class that references another class through a data member of a pointer type. The referenced class is, however, too simple to be considered as an Adaptee as it has very few members or too few methods of it are used by the Adapter. For example, the Adapter contains a String that holds the name of the current object, and one of the Adapter's methods needs the length of the String so it calls the corresponding method of String. Clearly, these candidates are not real Adapter Object patterns.

Candidates of the second group have an Adapter class that implements some kind of a collection data structure like a set, list or an iterator. These code fragments also have the structure of an Adapter Object design pattern but their purpose is obviously different, so they are not real patterns either.

After we had classified the candidates as true or false instances and the predictor values had been calculated, we investigated whether the actual predictor values support our assumptions about the predictors set forth in Section 4.2. Table 2 shows the predictor values for some typical candidates along with their manual classification results.

A1	A2	A3	A4	A5	A6	Classification
5	4	0	0	1	7	True
1	0	2	0	1	1	True
0	35	11	4	0.33	26	False
2	14	52	2	0	9	False

Table 2. Some predictor values for Adapter Object

Let us look at, say, the candidate in the first row of the table, which is a true pattern instance. We can see that more public methods in Adapter call the Adaptee (A1>A2), while there are no non-public methods that do not call it (A3=0). The values of A4, A5 and A6 also support our assumptions (all Adapter constructors take an Adaptee). Let us take the third row, which contains a false candidate, as another example. It can be seen that the relations between the values of A1, A2 and A3 are exactly the opposite of the true example, i. e. there are no many calls from Adapter to Adaptee. A5 and A6 support our assumptions as well. Based on this, we may safely assume that the learning methods probably discovered these relationships as well.

6.2 Strategy Candidates Investigation

First we will overview some interesting candidates we encountered during the investigation of Strategy instances. There was a class called SwModify, which behaved as a Context, while SwClient took the role of Strategy. The former had a method called Modify that would call the Modify method of the latter. Inherited (direct or indirect) classes of SwClient defined different Modify methods, so we classified this candidate as a true one. We also noticed that the number of ConcreteStrategy-s was quite high (65) that justified the usefulness of predictor S4.

The mentioned Strategy class, SwClient, appeared also in another candidate instance as the Strategy class, which we finally treated as a false one for the following reasons. The Context was represented by the class SwClientIter that communicated with SwClient through the AlgorithmInterface method called IsA. The ConcreteStrategy-s defined this method, but since its purpose was only RTTI (runtime type identification) and not that for a real algorithm (in other words, its S3 predictor was quite low), we classified this candidate as false.

There were also some candidates with only one ConcreteStrategy. We decided to classify these as false instances because, after investigating all other true instances, it was obvious that a real Strategy pattern should have several ConcreteStrategy-s otherwise its initial purpose is lost (the value of predictor S4 should be greater than one).

We also investigated the predictor values together with the classifications of Strategy in more depth in order to verify our initial assumptions about the predictors. Table 3 shows four example candidates with the predictor values and the classifications.

S1	S2	S3	S4	S5	S6	Classification
0	0	10.15	65	2	57	True
0	1	13.6	10	1	9	True
3	3	0	1	2	2	False
1	1	0	2	6	35	False

Table 3. Some predictor values for Strategy

The first row contains the predictor values for a true candidate. It can be clearly seen that the S1 and S2 predictors are low, while S3 and S4 are high, as expected. Predictor S5 is greater than 0 and S6 is smaller than S4, so this also supports being a true instance.

The last row represents a false candidate, where the S1 and S2 predictors are not zero. Furthermore S3 and S4 are very low, which suggests that this should be a false instance according to our assumptions. Finally, S6 is also much higher than S4, which further supports the belief that this is a false candidate. The manual classification was false, so our assumptions about the predictors were again correct.

In the next section, where we will show the actual results of learning efficiency, we will see that the learning methods successfully discovered these features of the predictors and incorporated them into their models.

6.3 Learning Efficiency

To assess the precision of the learning process we applied the method of three-fold cross-validation,[4] which means that we divided the predictor table file into three equal parts and performed the learning process three times. Each time we chose a different part for testing and the other two parts for learning.

We measured the learning precision in each case simply as the ratio of the number of correct decisions of the learning systems (compared to the manual classification) to the total number of instance candidates. Finally, we calculated the average and standard deviation (shown in parentheses) from these three testing results and got the values shown in Table 4.

Design Pattern	Decision Tree	Neural network
Adapter Object	66.70% (21.79%)	66.70% (23.22%)
Strategy	90.47% (4.13 %)	95.24% (4.12 %)

Table 4. Overall learning precision results

It can be seen that the two learning methods produced very similar results, however the precision was worse in the case of Adapter Object. This is probably due to two reasons. First, one of the three validation tests produced very bad results which worsened the overall percentages, and second, it seems that we have managed to find better predictors for Strategy than for Adapter Object.

However, the real importance of the achieved learning precision will be appreciated only by investigating how the application of machine learning improves the precision of the design pattern recognition. To do this we defined the following measures (see Tables 5–8):

- **Effectiveness** It measures the effectiveness of filtering out false hits, and is essentially the ratio of the number of correctly predicted false classifications by the decision model to the number of manually identified (observed) false hits of the basic method.
  (*Really false* / *False observed*)
- **Reliability** It shows to what extent the prediction of false hits was wrong, i. e. the ratio of the number of correctly predicted false classifications to the total number of false classifications.
  (*Really false* / *False predicted*)
- **Completeness** It measures how many of the manually identified (observed) true hits are correctly predicted by the learning method.
  (*Really true* / *True observed*)
- **Correctness** It shows the degree of correctly predicted true hits.
  (*Really true* / *True predicted*)

Tables 5–8 show these measures for the two learning methods. It may be concluded from the tables that the filtering effectiveness is really good with both learning methods and design patterns (85–94%), and that few true hits were wrongly predicted as false ones, so the reliability of filtering is also quite good.

[4] We did not have enough design pattern instance candidates to perform the usual ten-fold cross-validation method so we decided to divide the training set into three parts instead of ten.

Design Pattern	Total hits	False observed	False predicted	Really false	Effectiveness	Reliability
Adapter Object	84	59	71	51	86.44%	71.83%
Strategy	42	35	35	33	94.29%	94.29%

Table 5. Decision tree statistics for false hits

Design Pattern	Total hits	False observed	False predicted	Really false	Effectiveness	Reliability
Adapter Object	84	59	69	50	84.75%	72.46%
Strategy	42	35	33	33	94.29%	100%

Table 6. Neural network statistics for false hits

Design Pattern	Total hits	True observed	True predicted	Really true	Completeness	Correctness
Adapter Object	84	25	13	5	20.00%	38.46%
Strategy	42	7	7	5	71.43%	71.43%

Table 7. Decision tree statistics for true hits

Design Pattern	Total hits	True observed	True predicted	Really true	Completeness	Correctness
Adapter Object	84	25	15	6	24.00%	40.00%
Strategy	42	7	9	7	100%	77.78%

Table 8. Neural network statistics for true hits

As for the ability to predict true hits, the results are diverse. Strategy instances were fairly well predicted, but only one fourth to fifth of Adapter Object instances were found by the learning methods with modest correctness. We can explain this effect by the fact that the predictors were designed with the intent to filter out false pattern instances and not to support true ones.

7 Conclusion and Future Work

In this paper we presented an approach with which significant improvements in precision can be achieved in design pattern recognition compared to the usual structure matching-based methods. The main idea here was to employ machine learning methods in order to refine the results of the structure-based approaches.

Our goal was to filter out the false hits from the results provided by our structure-based pattern miner algorithm presented in [4]. In our experiments we achieved learning precisions of 67–95% and with the model obtained we could filter out 51 of the 59 false hits of the Adapter Object design pattern (out of a total of 84 hits) and 33 of the 35 false hits of the Strategy pattern (out of a total of 42 hits).

While our experiments showed that this approach has the potential to enhance current design pattern mining methods, further efforts should be made to develop the system. This includes calculating further/better predictors, supporting all the design patterns from [11] and training the tool on further software systems (e. g. on large open source software like the Mozilla internet suite).

Our method is purely static, which means that no runtime information is used. This makes it easy to use, because no test-runs have to be performed. But, of course, at runtime additional information (like execution history) can be obtained, so one possible direction for future enhancements would be to also incorporate dynamic (runtime) information in our design pattern miner tool.

References

[1] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, and N. Jussien. Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together. In *16th International Conference on Automated Software Engineering (ASE'01)*, pages 166–173. IEEE Computer Society, 2001.

[2] G. Antoniol, R. Fiutem, and L. Cristoforetti. Using Metrics to Identify Design Patterns in Object-Oriented Software. In *Proceedings of the Fifth International Symposium on Software Metrics (METRICS98)*, pages 23–34. IEEE Computer Society, Nov. 1998.

[3] A. Asencio, S. Cardman, D. Harris, and E. Laderman. Relating Expectations to Automatically Recovered

Design Patterns. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 87–96. IEEE Computer Society, 2002.

[4] Z. Balanyi and R. Ferenc. Mining Design Patterns from C++ Source Code. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003)*, pages 305–314. IEEE Computer Society, Sept. 2003.

[5] J. M. Bieman, D. Jain, and H. J. Yang. Design Patterns, Design Structure, and Program Changes: An Industrial Case Study. In *Proceedings International Conference on Software Maintance (ICSM 2001)*, pages 580–589. IEEE Computer Society, 2001.

[6] C. M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford., 1995.

[7] M. Campo, C. Marcos, and A. Ortigosa. Framework comprehension and design patterns: A reverse engineering approach. In *In Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, June 1997.

[8] R. Ferenc and Á. Beszédes. Data Exchange with the Columbus Schema for C++. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 59–66. IEEE Computer Society, Mar. 2002.

[9] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, Oct. 2002.

[10] R. Ferenc, J. Gustafsson, L. Müller, and J. Paakki. Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa. *Acta Cybernetica*, 15:669–682, 2002.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co, 1995.

[12] Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi. Fingerprinting Design Patterns. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 172–181. IEEE Computer Society, 2004.

[13] The Imagix Homepage. `http://www.imagix.com`.

[14] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-Based Reverse-Engineering of Design Components. In *The 21st International Conference on Software Engineering (ICSE'99)*, pages 226–235. IEEE Computer Society, 1999.

[15] J. Niere, M. Meyer, and L. Wendehals. User-driven adaption in rule-based pattern recognition. Technical Report tr-ri-04-249, University of Paderborn, Paderborn, Germany, June 2004.

[16] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A. Verkamo. Software metrics by architectural pattern mining. In *Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress)*, pages 325–332, 2000.

[17] J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[18] F. Shull, W. L. Melo, and V. R. Basili. An Inductive Method for Discovering Desing Patterns from Object-Oriented Software Systems. Technical report, 1996.

[19] M. Siff and T. Reps. Identifying modules via concept analysis. In *The International Conference on Software Maintance (ICSM'97)*, pages 170–179. IEEE Computer Society, Oct. 1997.

[20] The StarOffice Homepage. `http://www.sun.com/software/star`.

[21] P. Tonella and G. Antoniol. Object oriented design pattern inference. In *Proceedings of the International Conference on Software Maintenance (ICSM '99)*, pages 230–238, Washington, DC, USA, 1999. IEEE Computer Society.

[22] L. Wendehals. Improving design pattern instance recognition by dynamic analysis. In *Proceedings of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA*, May 2003.

[23] L. Wendehals. Specifying patterns for dynamic pattern instance recognition with uml 2.0 sequence diagrams. In *Proceedings of the 6th Workshop Software Reengineering (WSR2004)*, pages 63–64, May 2004.

[24] P. Wendorff. Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project. In *Fifth European Conference on Software Maintenance and Reengineering*, pages 77–84. IEEE Computer Society, 2001.