Automatic forecasting of design anti-patterns in software source code

Łukasz Puławski

Institute of Informatics, Warsaw University,
Banacha 2, 02-097 Warszawa, Poland,
Lukasz.Pulawski@mimuw.edu.pl

Abstract. The paper presents a framework for automatic inferring knowledge about reasons for the appearance of anti-patterns in the program source code during its development. Experiments carried out on histories of development of few open-source java projects shown that we can efficiently detect temporal patterns, which are indicators of likely appearance of future anti-pattern.
The approach presented in this paper uses expert knowledge (formal description of anti-patterns) to automatically produce extra knowledge (with machine learning algorithm) about the evolution of bad structures in the program source code. The research can be used to build scalable and adaptive tools, which warns development teams about the fact that system architecture is drifting in the wrong direction, before this is reported by typical static source code analysis tools.
Key words: Software evolution, temporal patterns, software design anti-pattern, machine learning

Introduction

Software development is a long-lasting process, which involves many developers. The main outcome of it is a system source code, which may consist of thousands of entities, such as files, classes or methods. Some of them may have certain bad properties, which make them more prone to defects, harder to understand or maintain. There are tools which can automatically asses entities and check if they are ill-structured. Such tools are of great help when planning refactoring or doing a code review. Their disadvantage is that such analysis can only be done post factum: once there is a problem in the software, the tool can tell where to find it. This paper presents a framework for the identification of a few well known defects in program source code structures, which addresses this problem. It is a semi-automated approach which detects frequent indicators of bad structures in the software code, before they actually appear.

On a high level, it can be seen as a framework which takes expert knowledge about bad design concepts as an input, analyses the evolution of software and, with usage of artificial intelligence algorithm, produces extra knowledge about how such ill-structured elements of source code evolve over time and where they come from.

Reminder of this paper

Section 1 provides a concise description of notions related to software development process, which are referred to in this research. Section 2 gives a short introduction to classification - a typical issue in machine learning, which is used in this research. Section 3 describes proposed framework in more detail. In sections 3.3 and 3.4 you will find a report about the data used in the experiments and selected results. Finally, sections 4, and 5 compare research presented herein with related approaches, and give some concluding comments and plans for the future work.

1 Software development

In a collaborative software development environments, where many developers are working on the same program for a long time, usually two important systems are used: *Source Code Management* system (*SCM* for short) and *Issue Tracker* (*IT*), in order to keep the process under control.

SCM is typically one central source code management server, which keeps the current version of the program source code. It allows developers to apply their changes to a common source base in a transactional manner. Such atomic changes are called *commits* or *check-ins*. Every check-in is stored in the SCM with information about the author, the time, a message and a list of modifications in the source code.

IT is a system which stores information about all tasks, called *issues*, done or planned during the development of the system. One special type of such issue is *bug*, which represents a defect found in the software. The tasks in IT have their lifecycle, which consist of several steps, such as creation, assignment to the responsible person, resolution of the problem and, eventually, closure of the issue. All such actions are recorded by IT. Therefore, one can treat IT as a log of history of the system development, because all tasks done in it must be recorded in detail by IT.

Similarly, SCM can be viewed at as a record of the same history, but from different perspective. It stores record of all changes done to the source code. It is common practice to put an identifier of a task from IT to a message of commit done to SCM. If this is followed, two different views on the system development history become "synchronized". Thus, on the one hand, when looking at a history of a file in SCM, one can check what tasks entailed a modification in the file. On the other hand, when looking at a task, one can check what modifications in the source code were necessary to complete it. The synchronization is an additional portion of information, which can be used to infer more knowledge about software evolution (e.g. one can check how many bugs were fixed in certain source files).

In the research presented herein I treat software development process as a temporal stream of events recorded by both SCM and IT, which are synchronised in the way described above.

1.1 Call graph

Formally, *call graph* is as directed multigraph $CG =< V, E >$, where V is a set of all subroutines in the program source code, and edge $< f, g >$ belongs to E iff f calls g. In object-oriented languages, subroutines are methods of classes. Therefore, the definition can be rephrased and expressed on class level: V is a set of all classes in the program source code and $< f, g >$ belongs to E iff f has a method that calls some method in g. Unless stated otherwise, this paper considers call graph on class level (i.e. according to the latter definition).

1.2 Inheritance tree

In programming languages with single inheritance[1], the *Inheritance tree* is a tree which contains all classes defined in the program source code. Each class is represented by a node, and the node of A is a parent of the node of B iff B inherits directly from A. Predicate $INH(a, b)$ denotes that class a is a subclass of class b.

1.3 Software metrics

Source code metrics are well-known tools for static code analysis. Formally, a source code metric is a function defined on a set of source code units, such as files, classes or methods, with numeric values. They measure the complexity of source code units and thus provide information about potentially ill-structured parts of the code, which may be error-prone or hard to maintain. The correlation between high (read: improper) values of source code metrics and the number of defects in the corresponding source code units has been widely analysed and proved to be true ([8], [11]). We will use the following notion: If c is a code entity and m is a metric, then $m(c)$ denotes the value of metric m for entity c. When evolution of c over time is considered, $m(c_{rev})$ denotes value of metric m on version of entity c from revision rev. When c is obvious from the context, these values are just written as m and m_{rev}.

In the research presented herein I used the following metrics: *Class data abstraction coupling* (denoted by Da), *Class fan-out complexity ($FanOut$)*, *Class fan-in complexity ($FanIn$)*, *Cyclomatic complexity ($Cycl$)*, *Number of effective lines class ($NCSS_c$)*, *Depth of inheritance tree (DIT)* and *Number of methods (NM)*. Detailed descriptions of these metrics can be found in [8], [11], [12], [15], [1] or [14].

1.4 Design (anti-)patterns

In software engineering, *design pattern* is a frequently used, universal resolution of commonly occurring problems in software design. This concept is not strictly

[1] This research is limited only to programs written in Java, which has only single inheritance.

formalised - it is rather an idea how to approach certain problems. However, in many cases it can be approximated in a formal model build from elements such as values of source code metrics and structures in program call graph and inheritance tree.

Similarly, *design anti-pattern* is a frequently used, wrong resolution of certain types of problems in software design, which has well-known disadvantages.

2 Classification

Classification is one of the problems in machine learning, in which artificial intelligence algorithm, given a description of an object (usually as a set of values of attributes), has to assign it to one of several possible categories (called *decision classes*). Typically, data for classification is represented in a *decision table*, in which rows represent *objects* (also called *instances*), columns represent attributes and each cell gives value of the attribute for a given object.

There are many classification algorithms, which will not be discussed here, because this research uses them as a tool to build description of patterns in the software evolution process. More detailed information can be found in [13]. However, a few important aspects must be mentioned. Typically, in order to have a good quality of classification, the decision table has to have certain properties:

- The set of objects must be large enough, because otherwise there is a risk of *overfitness* ([13]).
- It should be balanced. i.e. each decision class should be represented equally among all objects, at least approximately. Otherwise the algorithm could find rules describing one decision class, but not necessarily those which discern it from other classes.

3 Framework

This section shortly describes the algorithm used to automatically infer knowledge about temporal patterns which lead to the appearance of anti-patterns in the software code.

The idea behind the algorithm is depicted in diagram 1. There are two main sources of data. One is the history of software evolution stored in the synchronized SCM and IT. The other input is the human expert knowledge represented as a formal description of anti-patterns in terms of elements of call graph, inheritance tree, defect statistics and values of metrics. They are presented in table 1. The output of the algorithm is a formal description of rules which characterise early indicators of evolution which is likely to end up in an anti-pattern.

The algorithm works as follows: Single run of it is dedicated to analyse the evolution of one and only one type of anti-pattern. First, raw data from SCM and IT logs is transferred into a more convenient form: for each revision from SCM a complete call graph, inheritance tree, values of metrics for source code entities

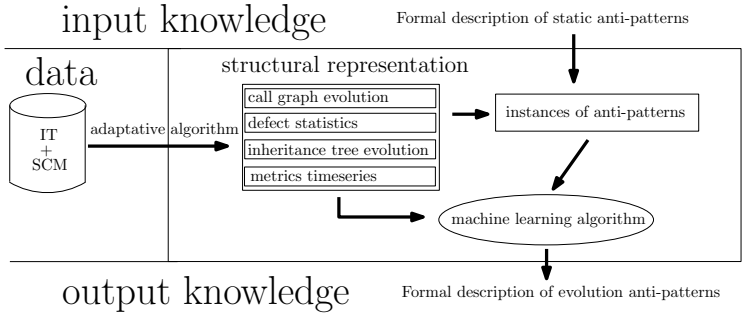

Fig. 1. Framework presented conceptually in the diagram

and statistics of resolved defects in it are computed.[2] This representation will be called *structural*. In the next step, all appearances of the analysed anti-pattern are identified in the most recent revision of the source code. Finally, a decision table is built, in which evolution of each identified instance of anti-pattern is represented by one row. This is done in order to run a classification algorithm, which outputs rules, that characterise the evolution of anti-pattern. A formal description of evolution anti-patterns is described in section 3.4.

3.1 Detailed description of the algorithm

Definitions This section defines notions used in the following paragraphs.

rev stands for the current, most recent revision of the analysed source code. For the following definitions we assume that revision rev is fixed.

Anti-pattern is a set of classes, which has certain properties that can be expressed in terms of the elements of structural representation. Detailed description of particular anti-patterns analysed in this research is given in section 3.4. For the following definitions we assume that there is a fixed instance of anti-pattern AP at revision rev.

rev_{start} is the first revision, in which AP can be observed. Obviously $rev_{start} \leq rev$.

rev_{before} is the last revision, in which AP is not observed and at least one of classes from AP is present in the SCM. If there is no such revision, then $rev_{before} \leftarrow 0$, in other cases $rev_{before} < rev_{start}$.

rev_{init} is the first revision, which is not greater than rev_{before} and in which either of the classes of AP were present in the SCM.

Evolution of AP, denoted by E_{AP} is a sequence of all revisions $\{rev_n\}_{n=\{1...k\}}$ in which at least one of the classes from AP was modified, such that $rev_1 = rev_{init}$, $rev_k = rev_{before}$ and $rev_{l-1} < rev_l$, for each l. Conceptually, it is a time

[2] In fact this can be implemented in a more efficient manner, if data is collected during software development. For details see chapter 3.2

period between rev_{init} and rev_{before} revisions, in which the source of all classes belonging to AP was evolving just before they become an instance of an anti-pattern. Note that in this research we only consider cases where $rev_{before} > 0$ and $rev_{init} < rev_{before}$, that is, where evolution of AP is a non-empty sequence of revisions. Figure 2 depicts all definitions given above.

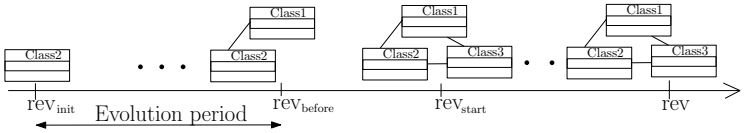

Fig. 2. Notions in pattern evolution

Algorithm All experiments, though aimed at detecting different anti-patterns, followed the same scheme. An approach for a single anti-pattern is described in meta-algorithm 1 and explained in more detail in the following paragraphs. The goal of it is to provide a decision table (see section 2), which can be analysed by classification algorithm in order to detect temporal patterns which frequently lead to the appearance of an anti-pattern in the source code. Conceptually, it provides a description of the whole evolution period of the anti-patterns, and encapsulates it in a single row of the decision table per instance of the anti-pattern.

Algorithm 1 GLOBAL META-ALGORITHM(rev)

Require: rev - revision of system source code
Ensure: DT - decision table for machine learning algorithm

1: $DT \leftarrow$ empty decision table ;
2: $CG \leftarrow$BUILD-CALL-GRAPH(rev);
3: $IT \leftarrow$ BUILD-INHERITANCE-TREE(rev);
4: $M \leftarrow$ COMPUTE-METRICS(rev);
5: $SAP \leftarrow$ FIND-ANTIPATTERNS(M, CG, IT);
6: **for all** $AP \in SAP$ **do**
7: $E_{AP} \leftarrow$ Evolution of AP {as described in section 3.1}
8: *positive_object* $\leftarrow$ DT-OBJ(E_{AP}, AP);
9: $k \leftarrow$ length of E_{AP};
10: $NON\_AP \leftarrow$ RANDOM-COLLABORATION(k);
11: $E_{NON\_AP} \leftarrow$ Evolution of NON_AP {as described in section 3.1}
12: *negative_object* $\leftarrow$ DT-OBJ(E_{NON_AP}, NON_AP)
13: Add *positive_object* and *negative_object* to DT
14: **end for**
15: **return** DT

In the first step (lines 2 - 4) the algorithm reads all files at a given revision from SCM and IT, and transforms it into structural representation. This allows to find occurrences of anti-patterns (line 5).

The next step (line 7), which is repeated for every identified instance of the anti-pattern, builds the history of all classes which comprise it. That is, it takes all revisions from the evolution period (from rev_{init} to rev_{before}) and for each revision it computes values of metrics for all classes. Additionally, it checks if either of the classes belonged to any other anti-pattern, different than AP. All data built in th is step is then used to build an object of decision table (line 8). This is, represented by routine DT-OBJ, described in section 3.1.

Lines 9 - 12 in the algorithm create another object in the decision table, which has similar properties to AP but is not an instance of the same anti-pattern. This is done by picking at random a set of classes (NON_AP), which has the following properties: 1) Power of NON_AP is equal to the power of AP, 2) each class in NON_AP has at least k revisions in its history, where k is the length of E_{AP}, 3) NON_AP is not an instance of the same anti-pattern as AP. The motivation for this part is to have a balanced decision table (see section 2). This is ensured because there are two decision classes ("Anti-pattern" and "Not-anti-pattern") and for every object belonging to the former (*positive_object*) there is one object which belongs to the latter (*negative_object*).

Creation of a row in decision table Routine DT-OBJ builds one object in the DT, based on information about the evolution of the anti-pattern AP (or random collaboration NON_AP) over period E. It works according to the following scheme:

- For each metric m, and every class $C \in AP$ (respectively: NON_AP) create the following attributes: $max_{rev \in E}\{m(C_{rev})\}$ (maximum value of metric), $avg_{rev \in E}\{m(C_{rev})\}$ (average value of metric), $max_{rev \in E}\{m(C_{rev})\} - min_{rev \in E}\{m(C_{rev}))\}$ (amplitude of metric). Since the number of classes in the analysed anti-pattern is fixed, this always gives 3 $\times$ *(number of metrics)* $\times$ *(number of classes)* numeric attributes.
- for every type of analysed anti-pattern (see table 1) different than AP, add one boolean attribute which indicates if at least one class in AP (NON_AP) belongs to it. The motivation for the introduction of these attributes is a hypothesis that certain anti-patterns tend to appear in the same area of code consecutively one after other.
- Eventually, give the object a decision attribute, which is *"Anti-pattern"* for AP and *''Not-anti-pattern"* for NON_AP.

3.2 Identification of anti-patterns

Identification of anti-pattern, represented by FIND-ANITPATTERNS routine is done by simple scanning structural representation for a set of classes with certain properties. The list of analysed anti-patterns together with the description of how they were identified is given in table 1. The algorithm to find occurrences of anti-patterns in the code has been taken from [17].

Table 1. List of analysed anti-patterns

Anti-pattern	Description	Formal description of anti-pattern
God object	Object which implements too much functionality	$(c) : NM(c) > 30 \wedge NCSS(c) > 1000$
Base bean	A class with utility methods used by its subclasses	$(c) : NM(c) > 30 \wedge \lvert\{(x, c) \in CG : INH(x, c)\}\rvert > 20$
YoYo	Too large inheritance tree of one class	$(c_1, c_2, \ldots c_6) : INH(c_i, c_{i+1}), i = 1 \ldots 5$
Buggy[3]	The class with many defects	Top 2% classes which contain most defects identified in IT. The anti-pattern consist of only this class

Scalability Since raw data used in this research has a significant size (thousands of source files, thousands of revisions) scalability of the framework is an important aspect. The algorithm to build a structural representation is designed in such a way, that it can build it adaptively. Note that software development process has good locality property: At one commit usually only a small number of files is modified. Therefore, in order to rebuild structural representation, source code parsing can be limited to only these classes which are defined in the modified files. Once this is done, any changes in the inheritance tree and related metrics (DIT) need to be applied only to classes which inherit from modified ones.

3.3 Data for experiments

Table 2 presents a list of particular datasets used in the experiments. Each row defines what portion of project development history was downloaded from respective JIRA (issue tracker) and SVN (source code management) servers and transformed into datasets used in the experiment.

3.4 Results

The decision table constructed in the way described above was analysed with an exhaustive algorithm in RSES system. It produces a rule-based classifier with the use of rough-sets methods (see [13]). The general concept of constructing rule-based classifiers in RSES is based on the extension of approximation spaces, as defined in e.g. [16]. Inferred rules were tested with a 5-fold cross validation

[3] This is not exactly an anti-pattern according to the definition given above. Nevertheless, this category of classes was also considered in the experiment to find out if the algorithm can identify frequent evolution patterns that make source code units contain many defects.

Table 2. Datasets used in the experiments

Dataset	JIRA	JIRA project	Issues range	SVN	approximate revisions range
Jboss	[9]	JBAS	6500 - 7700	[10]	85000-100000
Myfaces	[5]	MYFACES	1400 - 2100	[6]	440000-890000
Axis2	[5]	AXIS2	4400 - 4700	[6]	550000-950000
Derby	[5]	DERBY	2500 - 4500	[6]	520000-900000
Hadoop	[5]	HADOOP	5000 - 6800	[6]	730000-960000
Geronimo	[5]	GERONIMO	4500 - 5500	[6]	730000-980000
Struts1	[5]	STR	2200 - 3200	[6]	400000-750000

method. Standard voting was a strategy used to resolve conflicts when a new object was classified. For details about implementation of classification algorithms please see [3].

Interpretation of results Rules produced by the classification algorithm might be difficult to understand by an unexperienced person, because they are expressed in the attribute logic. However, certain human-readable interpretations can be derived from them. There are two types of attributes in the decision table: those representing aggregated values of metrics, and those related to existence of anti-pattern. As the latter are just a boolean attributes which represent appearance (or not) of a certain anti-pattern X, each can be expressed directly as ,,anti-pattern X was (not) found previously in this source unit". For metric-related attributes, examples of such interpretations are given in table 3

Table 3. Interpretation of evolution patterns

Conditions	Interpretation
$min(m)$ is low and both $max(m)$ and $m_{rev_{end}}$ are high and $avg(m)$ is low	,,Value of metric m was low, until it grown rapidly"
$avg(m)$ is low and $amp(m)$ is low	,,Value of metric m was constantly low"
$avg(m)$ is medium and $amp(m)$ is high	,,Value of metric m was changing in wide scope"

Please note that table 3 contains just examples and not a complete list. Additional rules can be easily deducted by analogy or duality. Note also that ,,low", ,,medium" and ,,high" terms used in the table are in fact numeric parameters of the experiment, which were given concrete values (per metric) when it was carried out.

Clearly, the interpretations given in table 3 are just approximations of sequential patterns which occur during the evolution period. Approximations, which might not always be true. This limitation can be overcame, when the proposed approach is implemented in a real software development environment, since data about values of metrics can be collected at every revision. Then it is obviously

very easy to say, with arbitrary confidence, if e.g. ,,value of the metric remains low". However, results of experiments show that even this very simple approximate model is sufficient to produce reasonable results.

Examples of anti-patterns and their representation This section gives a few interesting examples of knowledge which is taken from rules inferred on the data described in table 2. The machine learning algorithm actually outputs a formal description of detected evolution anti-patterns in a form of decision rules ([3]). In this paragraph, these rules have been translated into natural language for better comprehensibility.

Anti-pattern Base bean may evolve to God object. The Rule is: "Evolution period of God object contains revisions which satisfy formal description of Base bean". The opposite direction (evolution from God object to Base bean) is less likely.

In the YoYo pattern it is usually the case that classes at the top and at the botton of inheritance tree are changed most (c_1 and c_6 have largest $NCSS$ metric amplitude), whereas intermediary classes tend to be stable in terms of size (again, amplitude of $NCSS$).

The YoYo pattern frequently evolves in such a way that it starts with a base class, which becomes large and relatively complex (large maximum and amplitude of $NCSS$ and $Cycl$ metrics for c_1), and it gradually has more and more derived classes, which are usually compact and not complex (maximum and amplitude of $NCSS$ and $Cycl$ for classes $c_1, \ldots, c_4$ is limited).

Base beans and God classes are frequently among the most defect-prone classes in the system (Evolution period of anti-pattern "Buggy" contains versions which satisfy formal definition of either Base bean or God class).

4 Related work

Temporal patterns in software evolution have been studied by many researchers. A common approach for the study focused on case studies done by human expert ([19], [2]). In some cases observation and characterisation of these patterns was made by human based on visualisation of the evolution depicted in special diagrams ([7] , [4]).The difference of approach presented is that it focuses on an automatic inferring of knowledge about temporal patterns in the software development process. A stress is put on the attempt to take "intelligent" work from human expert to an artificial intelligence algorithm to the maximum possible extent.

In [18] authors also address problem of mining temporal patterns in the software development log. The difference is that the unit of time presented in this paper is a release (version) of software. My research focuses on commit-time patterns and therefore can be used to react almost immediately to appearance of bad structures, especially that model can be built in an adaptive way.

5 Conclusion

The paper presents a framework, which has been experimentally proven to be efficient in automatically discovering new knowledge about evolution patterns in the software development process. Note that all examples given in section 3.4 are not human-driven case studies, but interpretations of rules automatically produced by the computer. It means that the concept proposed herein can help software developers and architects in non-trivial tasks related to detecting bad quality indicators. The concept of detecting static anti-patterns is based on existing research [17], but the research presented herein contains two novel approaches: the adaptive algorithm for constructing a *Structural representation* and a framework for automatic detection of evolution patterns. This gives opportunity to use the approach in large scale software systems.

6 Future work

This paper reports a preliminary results of predicting a few simple anti-patterns in selected open-source projects, with the proposed framework. Further improvements are foreseen. On one hand I plan to apply this model to detect more complex anti-patterns. On the other hand, I want to develop a more sophisticated method of constructing the decision table, so that it contains more information about temporal patterns. Among other things, I want to enrich it with information about duration of evolution period as well as duration and frequency of appearances of other anti-patterns in it.

In the current approach the input knowledge, represented as a formal description of anti-patterns, has to be provided by a domain expert. In order to reduce his involvement, I want to check if important features of the evolution period can be extracted automatically by the computer. This could make the whole framework more autonomous and enable it to adapt to different environments (e.g. different programming styles). Similarly, interpretations of results produced by the machine learning algorithm and their transformation to natural language can potentially be automated.

References

1. Checkstyle tool home page. http://checkstyle.sourceforge.net/config_metrics.html.
2. T. Apiwattanapong, A. Orso, and M. J. Harrold. A Differencing Algorithm for Object-Oriented Programs. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.
3. J. G. Bazan, M. S. Szczuka, and J. Wroblewski. A new version of rough set exploration system. In J. J. Alpigini, J. F. Peters, J. Skowronek, and N. Zhong, editors, *Rough Sets and Current Trends in Computing*, volume 2475 of *Lecture Notes in Computer Science*, pages 397–404. Springer, 2002.

4. M. D'Ambros, M. Lanza, and M. Lungu. Visualizing Co-Change Information with the Evolution Radar. *IEEE Transactions on Software Engineering*, 35(5):720–735, Sept. 2009.
5. A. foundation. Issue tracker. https://issues.apache.org/jira/.
6. A. foundation. Scm. http://svn.apache.org/repos/asf/.
7. T. Girba and M. Lanza. Visualizing and Characterizing the Evolution of Class Hierarchies, 2004.
8. M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.
9. JBoss. Issue tracker. https://jira.jboss.org/browse/JBAS.
10. JBoss. Scm. http://anonsvn.jboss.org/repos/jbossas/.
11. T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, 1976.
12. B. A. Nejmeh. Npath: a measure of execution path complexity and its applications. *Commun. ACM*, 31(2):188–200, 1988.
13. Z. Pawlak and A. Skowron. Rudiments of rough sets. *Information Sciences*, 177(1):3–27, 2007.
14. Ł. Puławski. Software Defect Prediction Based on Source Code Metrics Time Series. In J. Peters, A. Skowron, C.-C. Chan, J. Grzymala-Busse, and W. Ziarko, editors, *Transactions on Rough Sets XIII*, volume 6499 of *Lecture Notes in Computer Science*, chapter 7, pages 104–120. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2011.
15. B. Ramamurthy and A. Melton. A synthesis of software science measures and the cyclomatic number. *IEEE Trans. Softw. Eng.*, 14(8):1116–1121, 1988.
16. A. Skowron, J. Stepaniuk, and R. W. Swiniarski. Approximation spaces in rough-granular computing. *Fundamenta Informaticae*, 100(1-4):141–157, 2010.
17. K. Stencel and P. Wegrzynowicz. Detection of Diverse Design Pattern Variants. In *2008 15th Asia-Pacific Software Engineering Conference*, pages 25–32. IEEE, Dec. 2008.
18. Q. Tu and M. W. Godfrey. An Integrated Approach for Studying Architectural Evolution. In *Proceedings of the 10th International Workshop on Program Comprehension*, IWPC '02, Washington, DC, USA, 2002. IEEE Computer Society.
19. Z. Xing and E. Stroulia. Data-mining in Support of Detecting Class Co-evolution. In F. Maurer and G. Ruhe, editors, *SEKE*, pages 123–128, 2004.