Proceedings of the of the 17th Conference on Computer Science and Intelligence Systems pp. 521–528
DOI: 10.15439/2022F211
ISSN 2300-5963 ACSIS, Vol. 30

Improvement of design anti-pattern detection with spatio-temporal rules in the software development process

Łukasz Puławski
Faculty of Mathematics, Informatics and Mechanics, University of Warsaw
Email: lpulawski@mail.mimuw.edu.pl

***Abstract*—In [1] we presented a framework for mining spatio-temporal rules in the software development process. The rules are based on specific relations between structures of the source code which relate both to spatial (e.g. a direct call between methods of two classes) and temporal dependencies (e.g. one class introduced into the source code before the other) observed in the process. To some extent, spatio-temporal rules allow us to predict where and when certain design anti-patterns will appear in the source code of a software system. This paper presents how, with slight modifications, such framework can be used to improve the quality of detecting a few popular design anti-patterns, such as Blob, Swiss Army Knife, YoYo or Brain Class. In the proposed method, we not only check the structure of a piece of the source code, but we also analyse its spatio-temporal relations. Only on the basis of the two analyses can we decide if the given piece of code is an anti-pattern. Experimental validation shows that the addition of spatio-temporal perspective improves detection of anti-patterns by 4% in terms of F-measure.**

I. Introduction

DESIGN anti-pattern is a commonly used, bad solution for a recurring problem in software design. Software developers, when faced with a common design problem, tend to reinvent the solutions that are well-known for their bad properties and widely discussed in available literature (see [2]). This phenomenon is definitely due to many sophisticated reasons, which are discussed in a variety of scientific and popular publications (see [3]). The following paragraphs give a few examples of design-anti-patterns.

Base bean is an anti-pattern in object-oriented design, where the base class is a collection of numerous utility methods used by its subclasses. Such a design breaks some fundamental concepts of object-oriented programming: The relation between subclass and superclass does not resemble the actual domain model, the superclass has many responsibilities and it usually does not store any state useful for subclasses.

Brain class and God Class are similar anti-patterns that refer to classes that provide too much complexity and tend to centralize logic of some area of the system. The difference between God class and Brain class is that the former is a large controller class that depends on data from the surrounding classes, whereas the latter does not use the data from other classes, tends to be more cohesive and encapsulates the logic in its own complex methods (see [4], [5], [6], [7]).

Swiss Army Knife, (abbreviated as *SAK*) is an excessively complex class with numerous unrelated utility methods. It tends to appear when the creator attempts to provide a routine for all possible uses of the class or make a single class serve many complex unrelated functions (see [8], [9] and [6]).

YoYo is an anti-pattern in which the flow control is scattered over complicated inheritance structure, so that in order to understand the algorithm in the source code, one has to switch between many classes within a common inheritance tree (see [10], [11], [6]).

Design anti-patterns make the software more complex, harder to maintain and defect-prone (see [12], [13], [14], [15]). This is why their detection is a primary concern in software engineering.

II. Design anti-pattern detection

The objective of design anti-patterns detection is to provide an automated method for the discovery of fragments of the program source code which constitute a design anti-pattern. To make this task more formal we will use graph-theoretical terms.

A. Software as a graph

We can treat the source code of the system written in Java as a multigraph called $software\ snapshot$ according to the following rules: The nodes of the multigraph are all the *source code entities*, namely: packages, interfaces, classes, fields, methods (including constructors). They can be connected by labeled edges according to the following rules:

- There is an edge (e_1, e_2) labeled 'contain' iff the source code of the entity represented by e_2 is contained in the source code of the entity represented by e_1 (we will assume that classes and interfaces are contained in packages),
- There is an edge (e, c) labeled 'variable' iff the body of the entity represented by e declares at least one variable of the type represented by class c. Each such declaration corresponds to a single edge.
- There is an edge (m, c) labeled 'parameter' iff the method represented by entity m declares at least one formal parameter of the type represented by c. Each such declaration corresponds to a single edge.
- There is an edge (c_1, c_2) labeled 'extend' iff the class represented by c_1 is a direct subclass of the class represented by c_2 or the class represented by c_1 is an implementation

of the interface represented by c_2. Each such class extension or interface implementation corresponds to a single edge.
- There is an edge (e, m) labeled 'call' iff the body of the entity represented by e contains at least one call of a method represented by the method entity m. Each such call corresponds to a single edge.
- There is an edge (e, f) labeled 'refer' iff the body of the entity represented by e contains at least one reference to the field represented by the field entity f. Each such reference corresponds to a single edge.
- There is an edge (f, c) labeled 'type' iff c represents a class that is a declared type of the field represented by f or a declared return type of a method declared by f.

Additionally, each node of the graph may be described by a set of applicable *software metrics* that measure its complexity (see [16]). For greater consistency we will assume that the values of metrics form a vector of additional labels of the node. Consequently, we can treat a software snapshot as a node-labeled and edge-labeled multigraph. This allows us to remain in the graph-theoretical domain. The list of metrics that are used is given in the following subsection:

B. Software metrics

- **Data abstraction coupling**
  This metric is applicable for class entities and measures how many instances of other classes are instantiated within the source code of a given class.
- **Fan out**
  This metric is similar to *Data abstraction coupling*, which measures the number of classes a given class depends on.
- **Cyclomatic complexity** and **NPath complexity**
  These two metrics measure the complexity of a code block. Cyclomatic complexity, based on the classic work [17], denotes the number of decision point instructions within the body block increased by 1. The NPath complexity (see [18]) denotes the theoretical maximum number of different acyclic execution paths that could go through the code block.
- **NCSS - the number of lines of the source code in each entity (file, class, method)**
  This simple metric measures how many lines of code a given file, class or method take. Technically, the evaluations do not count empty lines or lines with comments, so that it approximates the actual size of the respective source code fragment.
- **Lack of cohesion of methods (LCOM1, LCOM2, LCOM3, LCOM4, TCC)**
  LCOM is a suite of software metrics that evaluate the design of a given class by quantitative analysis of relations between its methods and attributes (see [19]).
  LCOM1 is defined as the difference between the powers of two sets: the set of all pairs of different methods that use a non-empty disjoint set of class attributes and the set of all pairs of different methods that use at least one attribute altogether. If the result is negative, the value of this metric is set to 0.
  LCOM2 and LCOM3 metrics are defined for the class and the formulae to evaluate them, are based on the following notions: m - the number of methods in a class, A - the set of attributes of a class, m_a - the number of methods that access attribute a:

$$LCOM2 = 1 - \frac{\Sigma_{a \in A} m_a}{m * |A|}$$

$$LCOM3 = \frac{m - \frac{1}{|A|}\Sigma_{a \in A} m_a}{m - 1}$$

  LCOM2 corresponds to the fraction of methods that do not access a specific attribute normalized over all attributes. LCOM3 is similarly normalized with respect to attributes and methods and its value may range from 0 to 2.
  LCOM4 (see [20]) is expressed in terms of a graph of inter-method dependencies. We say that two methods are dependent iff one of them calls the other one or there is at least one attribute used by both methods. The number of connected components in such a graph is the value of LCOM4. Instead of looking at relations between methods and attributes within a single class, we can in a similar manner measure the relation between methods of a given class, with the use of Tight Class Cohesion (TCC) metric (see [21]). TCC is defined as the number of pairs of methods that invoke one another divided by the number of all such pairs.
- **Depth of inheritance tree (DIT)**
  This metric is applicable to classes only. It measures the number of nodes on the path in the inheritance tree from the node representing the `java.lang.Object` class to the node representing the given class. Large value of this metric indicates a deep inheritance tree, which might indicate the presence of a *Yo-yo* design anti-pattern.
- **Fan in (FI)**
  A metric dual to the *Fan out*. It measures the number of other classes that depend on a given class. The greater the value, the more likely it is that a change in the class will affect other fragments of the software source code.

In this context we can formalize the problem of detecting design anti-patterns as the problem of finding all such subgraphs of the software snapshot that correspond to instances of these anti-patterns. For practical reasons we will only consider subgraphs that satisfy the following Definition 1:

Definition 1 (containment-completeness): Let $SSn = (V, E)$ be a software snapshot, where V is a set of nodes and E is a multi-set of labeled edges. We will say that subgraph $g = (V_g, E_g)$ of SSn is $containment - complete$, if for any $n_1 \in V_g$, if there is a node $n_2 \in V$ such that n_2 is connected by 'contain' edge with n_1 in SSn then $n_2 \in V_g$.

We apply this constraint on the subgraphs, since we want the analysed subgraphs to resemble a consistent fragment of the source code with its natural hierarchical structure. For

example, if we take a subgraph with a node that corresponds to a class, we also want methods and fields of this class to be part of the subgraph. Therefore, in all the following considerations a *subgraph* always means a *containment-complete subgraph*.

C. Detectors of design anti-patterns

The following paragraphs provide a semi-formal description of method of detecting specific design anti-patterns used in this research. Each detection strategy is derived from existing research mentioned in the respective subsections below, but they are adapted to the graph-theoretical model described in Section II-A. In order to enhance comprehension, detection methods are described in mixed graph-theory and software-engineering terms. Each description can be translated to purely graph-theoretical terms so that for any subgraph of a software snapshot we can always tell if it satisfies the conditions described below. A exemplary rationale on how the conceptual definition can be translated to graph-theoretic model is given for the first detector only. Similar reasoning for other types of anti-patterns can be found in the articles referred to in the respective following subsections.

1) Swiss army knife: A *Swiss Army Knife* (*SAK* for short), is an excessively complex class interface. It is present when e.g. the creator attempts to provide a method for all possible uses of the class or make a single class serve many complex unrelated functions. Methods described or referenced in [8], [9], [6] and [22] provide a semi-formal description of the pattern as *a class with many unrelated methods with high complexity which implements many interfaces*. Clearly, this can be translated to the graph-theoretical language in the context of a software snapshot: a class with many interfaces corresponds to each node such that paths which start from it and contain edges of type 'extend' and 'implement' reach many nodes which represent interface entity. A complex method is simply a method with high values of the complexity metrics such as *NPath complexity* or *cyclomatic complexity*. Additionally, if a class is intended to serve many purposes, one can expect that it has methods that are being called by many other classes. This conceptual description can be translated in the formal graph-theoretical definition described below:

Definition 2 (foreign call): Let c_1 and c_2 be two classes such that they are not connected by a path build from edges labeled 'extend'. Every call from a method contained in c_2 is a *foreign call* for class c_1.

Swiss Army Knife is each class c that satisfies the following conditions:

- c has more than 6 methods,
- the value of metric LOC for c exceeds 150,
- the sum of cyclomatic complexity of methods contained in c exceeds 30,
- the sum of NPath complexity of methods contained in c exceeds 120,
- the number of non-trivial methods contained in c multiplied by the average NPath complexity of such methods exceeds 160,
- the number of methods called by a foreign call exceeds 2,
- the number of foreign calls exceeds 7.

2) Anemic entities: Anemic entities are classes which only store data and do not provide any functionality (see [23], [24]). A straightforward, naive approach for detecting this pattern is to take classes which have:

- many fields,
- only accessor methods (i.e. methods with a single line of code, which refer to only a single field and have cyclomatic and NPath complexity equal to one),
- default and possibly an initializing constructor.

This heuristic turns out to be inaccurate. Therefore, we need to define two notions: an *effectively trivial method* and a *complex constructor*.

A method m is *effectively trivial* if :

- The class c in which m is contained has field f of type t such that m refers to f and either m has 1 argument of type t and $void$ return type or it has 0 arguments and return type t,
- m has at most 5 lines of code,
- m has cyclomatic complexity not greater than 3.

The constructor con contained in c is complex if:

- The number of arguments of con exceeds the number of fields contained in c,
- the lines of code in con exceeds 150% of the number of fields contained in c,
- the cyclomatic complexity of con exceeds 150% of the number of fields contained in c.

This allows us to provide a formal definition for anemic entity: A class is an *Anemic Entity* iff:

- it has more than 8 fields,
- it has more than 8 methods,
- all methods but one are trivial or *effectively trivial*,
- there are no complex constructors contained in c,
- all subclasses of c satisfy the above four conditions.

3) Blob (also known as God Class): (see [4], [5], [6], [7]). Entity c is considered a Blob iff:

- c calls more than 7 effectively trivial methods of other classes,
- proportion of the number of effectively trivial methods of other classes called by c to the number of other methods of other classes called by c exceeds 0.6,
- the sum of cyclomatic complexity of its non-trivial methods exceeds 55,
- the value of the metric TCC does not exceed 0.3.

4) Brain class: (see [4], [5], [6], [7]). Entity c is considered brain class iff

- c is not a God Class, as defined in the preceding subsection,
- c has more than 2 non-trivial methods with more than 4 outgoing edges of type 'call' and more than 15 lines of code (controller methods),
- c calls more than 5 trivial methods of other classes,

- proportion of the number of trivial methods of other classes called by c to the number of non-trivial methods of other classes called by c does not exceed 0.6,
- the number of calls to trivial methods divided by the number of lines of the source of c is smaller than 0.2,
- the value of tight class cohesion metric for c does not exceed 0.5,
- One of the following conditions is true:
  i sum of cyclomatic complexity of methods contained in c exceeds 50 and the value of NCSS metric for c exceeds 400,
  ii sum of cyclomatic complexity of methods contained in c exceeds 90 and the value of NCSS metric for c exceeds 50,

5) Base Bean: (see [25]) Base Bean is a class which only provides utility methods for its subclasses.

A method m contained in class c is an *utility method* iff:

- m is neither a constructor nor a trivial method,
- m does not refer to any field contained in c, nor to a field in any direct or indirect superclass of c,
- there is no path that connects m with field f contained in c or one of its direct or indirect superclasses, such that the last edge on this path has a label 'refer' and all other edges have a label 'call'.

Conceptually, a utility method is a non-trivial method that does not modify the state nor does it orchestrate other methods contained in the class it is defined in or any of its ancestors and is used only by its descendants.

A class c is *Base Bean* iff:

- it has more than 2 utility methods,
- c has at least 5 direct or indirect subclasses,
- the number of incoming edges of type 'call' from the hierarchy of c to utility methods contained in c exceeds 2.

6) Yo-yo: (see [10], [11], [6]). A containment-complete sub-graph induced by nodes $Y = \{e_1, \ldots, e_n\}$ is a *YoYo* iff:

- Each pair $(e_i, e_j) \in Y \times Y$ is connected by a path constructed from edges of type 'extend', where each edge is treated as undirected,
- the longest path between any two nodes from Y constructed from such edges exceeds 5,
- the number of edges (m_1, m_2) with label 'call' or 'refer' such that m_2 is not a trivial method and there are edges $(m_1, e_i), (m_2, e_j)$ with label 'contain', $i \neq j$ exceeds 5.
- there is no super-set of nodes $Y' \supseteq Y$ such that graph induced by Y' satisfies the above three conditions.

7) Data Clumps: (see [15], [26], [27], [28]) a *Data Clump* is an anti-pattern that occurs when a group of data items are being passed together in the source code. This informal definition can be rephrased formally:

Let $parameters(m)$ denote set of entities connected with m by an edge labeled 'parameter'. A set of method entities $M = \{m_1, \ldots, m_n\}$ is a *Data Clump* iff:

- n exceeds 3,
- $|parameters(m_i)|$ exceeds 3 for each i,
- for each pair (m_i, m_j) such that $i \neq j$ and m_i and m_j are connected by 'call' edge, $|parameters(m_i) \cap parameters(m_j)| = min(|parameters(m_i)|, |parameters(m_j)|)$,
- there is no such superset of M that satisfies the above conditions.

8) Circular dependency: Circular dependency is a relation between two or more software entities transitively contained in different packages which either call each other directly or indirectly to function properly. We can translate this into graph theoretical terms:

A pair of classes (c_1, c_2) forms a circular dependency iff:

- there exist two different packages p_1, p_2 such that there are edges (c_1, p_1) (c_2, p_2) with label 'contain',
- there are two methods m_1, m_2, such that there are edges (m_1, c_1) and (m_2, c_2) with label 'contain',
- there is a path build only from edges labeled 'call' from m_1 to m_2 and another such path from m_2 to m_1.

9) Detection quality: The detection quality of purely static methods of identification of design anti-patterns described in the preceding subsections is presented in Table I.

	SAK	Bl	DC	BB
Argo Uml	0.78/1.0	0.90/0.76	N/A	0.71/0.88
Elasticsearch	0.78/0.99	0.83/0.9	0.99/0.96	0.71/0.88
JHotDraw	1.0 /0.91	1.0/1.0	0.28/0.0	N/A
Lucene	0.86/1.0	0.88/0.9	N/A	0.97/1.0
Struts	0.99/1.0	N/A	0.98/0.1	N/A
Wildfly	0.94/1.0	0.92/1.0	0.99/1.0	1.0/1.0
Xerces	0.8/0.89	0.91/0.69	0.99/0.84	N/A

	BC	YY	AE
Argo Uml	N/A	1.0/1.0	1.0/1.0
Elasticsearch	0.87/0.84	0.98/1.0	1.0/1.0
JHotDraw	0.0/0.0	N/A	N/A
Lucene	0.95/0.78	1.0/1.0	N/A
Struts	N/A	N/A	N/A
Wildfly	N/A	N/A	1.0/1.0
Xerces	N/A	0.98/1.0	N/A

Table I

THE TABLE SHOWS THE QUALITY OF DETECTION OF INSTANCES OF THE DESIGN ANTI-PATTERNS. THE COLUMNS CORRESPOND TO A RANGE OF ANTI-PATTERN TYPES WHEREAS THE ROWS PRESENT DIFFERENT SOFTWARE SYSTEMS. THE DETECTION METHODS ARE DESCRIBED IN SUBSECTIONS II-C1–II-C8 ABOVE. EACH CELL CONTAINS TWO NUMBERS: PRECISION/RECALL. SAK = SWISS ARMY KNIFE, BL = BLOB, DC = DATA CLUMPS, BB = BASE BEAN, BC = BRAIN CLASS, YY = YOYO, AE = ANEMIC ENTITY.

III. SPATIAL RELATIONS

Since design anti-patterns are subgraphs of one common graph, we can introduce the notion of distance between two patterns defined as the length of the shortest path that connects nodes from these subgraphs:

Definition 3 (closeness and remoteness of patterns): Let $PI_1 = (V_1, E_1)$ and $PI_2 = (V_2, E_2)$ be subgraphs of software snapshot SSn. We will say that

PI_1 and PI_2 are *d-distance-close* iff $d(PI_1, PI_2) \leq d$, where $d(PI_1, PI_2, SSn) = min_{v_1 \in V_1, v_2 \in V_2} dist(v_1, v_2, SSn)$ where $dist(a, b, G)$ is the distance between vertices a and b measured as the shortest path between them in the multigraph G treated as undirected graph.[1]

[1]This makes dist symmetric.

Similarly: PI_1 and PI_2 are *d-distance-remote* iff $d(PI_1, PI_2) > d$.

IV. Software evolution

Usually software development is done in the *source code management* system that allows us to track all changes done to the source code. Each individual modification of the source code is called *commit* and is identified by unique number called *revision*. Commit has precise date, author and a set of modifications to the source code files. If we take the commits from a single main development branch, we can order them linearly according to the commit date. The code at each revision has a corresponding software snapshot, thus we can treat linearly-ordered sequence of such snapshots as a model of *software evolution*.

One specific design anti-pattern can be observed at multiple revisions. The set of all such revisions will be called the *lifespan* of a pattern. Clearly, the lifespan can be divided into intervals of maximum lengths such that the corresponding pattern instance is not observed in the revision that directly precedes the left end of this interval nor is it observed in the revision that directly follows the right end of this interval. Each such interval will be called *occurrence* of the pattern. Please note that each pattern instance can potentially have more than one occurrence, as e.g. a certain software structure can be removed and then added again to the source code.

V. Spatio-temporal relations

If we take two different patterns PI_1 and PI_2 and their two occurrences $l_1 = (l^1_{start}, l^1_{end})$ and $l_2 = (l^2_{start}, l^2_{end})$, we can tell the *temporal* relation between l_1 and l_2 (e.g. l_1 may directly precede l_2 when $l^1_{end} = l^2_{start}$). In order to model the temporal relations we use Allens interval algebra in this research (see [29]), which introduces 13 different possible relations which comprise equality and 6 pairs of invertible relations.

We will say that Allens relation between l_1 and l_2 defined above is *non-inverted* iff $l^1_{start} < l^2_{start} \vee (l^1_{start} = l^2_{start} \wedge l^1_{end} < l^2_{end})$. Conceptually it means that l_1 *takes place before* some non-degenerated sub-interval of l_2. In other words, l_2 will last for some time after l_1 has started. The non-inverted relations of these pairs are given in the following list:

1) l_1 *takes place before* l_2 if there exists a revision s such that $(l^1_{end} < s < l^2_{start})$
2) l_1 *meets* l_2 $(l^1_{end} = l^2_{start})$
3) l_1 *overlaps* l_2 $(l^1_{start} < l^2_{start} < l^1_{end} < l^2_{end})$
4) l_1 *starts* l_2 $(l^1_{start} = l^2_{start} \wedge l^1_{end} < l^2_{end})$
5) l_1 *contains* l_2 $(l^1_{start} < l^2_{start} < l^2_{end} < l^1_{end})$
6) l_1 *is finished by* l_2 $(l^1_{start} < l^2_{start} \wedge l^2_{end} = l^1_{end})$

Each Allens operator A has its inversion A^{-1} (which is also an Allens operator) defined by: xAy iff $yA^{-1}x$.

Let $0 < d_c < d_r$ be two fixed natural numbers which we will associate with d_c-distance-closeness and d_r-distance-remoteness relation respectively. If l_1 and l_2 are in A Allen relation and at some revision graph induced by nodes of PI_1 is d_c-distance-close to graph induced by nodes of PI_2, then we will say that these two occurrences are in A-d_c-distance-closeness *spatio-temporal relation*. If these graphs are d_r-distance-remote at all revisions we will say that they are in A-d_r-distance-remoteness spatio-temporal relation. Please note that the above definitions are also valid if PI_1 and PI_2 are never observed together at a single revision.

If we take a single occurrence l_1 of some anti-pattern PI_1 ($[l_1, PI_1]$), we can tell all its spatio-temporal relations to all other occurrences of other anti-patterns. Each such relation can be characterized by three arguments:

- T - the type of other anti-pattern ($T \in \{$BLOB, SAK, Base Bean, YOYO, Brain Class, Data Clump, Anemic Entity, Circular Dependency$\}$),
- A - the non-inverted Allen algebra relation ($A \in \{$ takes place before, meets, overlaps, starts, contains, is finished by$\}$) and
- $s \in \{remote, close\}$ - which determines if we are talking about A-d_c-distance-closeness or A-d_r-distance-remoteness spatio-temporal relation.

Therefore, for each triplet (T, A, s) we can tell how many respective spatio-temporal relations to $[l_1, PI_1]$ exist in the software evolution. This yields a vector in $\mathcal{N}^{(8\times6\times2)}$ space which provides information about the number of all spatio-temporal relations of occurrence $[l_1, PI_1]$ in the entire evolution. The dimension of this space is related to the Cartesian product of:

- all types of anti-patterns described in Section II (8),
- the number of non-inverted Allen relations (6) and
- the number of types of different spatial relations from Definition 3 (2).

Consequently, the occurrence of an anti-pattern is described by a vector of 96 natural numbers.

Please note that the notions of closeness and remoteness from Definition 3, as well as the notions of lifespan, occurrence and spatio-temporal relations, are defined in such a way that they are also applicable to any sub-graph that can be observed in snapshots of the software evolution. Thus, we can compute the aforementioned 96 attributes for any occurrence of such a subgraph.

In ([1]) we argued that such a vector appears to be very specific for occurrences of design anti-pattens. This phenomenon may be interpreted as a tendency for certain design anti-patterns to appear close to each other (spatial relation) and one after another (temporal relation). They can therefore be used to predict areas in the source code, where design anti-pattern may appear in the future. In this research we will use similar framework to improve detection quality for detectors described in Section II.

A. Spatio-temporal rules

We will describe a method of mining *spatio-temporal rules* which allows us to reason about spatio-temporal relations in the entire software evolution. We will construct these rules by applying a rule-based machine-learning classification algorithm on specially prepared decision table. The following paragraphs describe how this table is constructed.

In Section V we argued that for any occurrence of design anti-pattern and any occurrence of any other subgraph of software snapshot we can compute a vector of 96 attributes that describe its spatio-temporal relations. For each occurrence of design anti-pattern PI of type t (e.g. Blob) we will insert one row to the decision table, with 96 conditional attributes and decision=t. We will call this a positive row for t (e.g. positive row for Blob). To balance this we will pick a random subgraph that does not correspond to anti-pattern t and has the same number of nodes as PI and insert it into decision table with 96 conditional attributes computed likewise. For such a row we will set decision=NOT_t (e.g. NOT_Blob). We will call such a row a negative row for t (e.g. negative row for Blob).

The decision table constructed according to the above description has 97 columns and the number of rows is twice the number of occurrences of all design anti-patterns in the entire software evolution. We can partition this table into smaller tables by selecting only positive and negative rows for only single type t of anti-patterns (e.g we only take rows with decision Blob and NOT_Blob). Each such sub-table is in fact a perfectly balanced binary decision table that can be used to train a machine-learning classification algorithm that produces a classifier in the form of a set of classification rules. We will call these rules *spatio-temporal rules* for t and the classifier will be called *spatio-temporal classier for* t. Technically, the classifier, given a vector of 96 natural numbers, outputs either t or NOT_t.

If we take occurrence l of some subgraph g in the software evolution we can compute 96 attributes for it and apply a spatio-temporal classifier on such a vector. Conceptually, the classifier for type t can tell if the given graph occurrence resembles a design anti-pattern t occurrence in terms of its spatio-temporal relations. We can use this observation to introduce an improved spatio-temporal detector for t. This concept is described in the following section.

VI. Spatio-temporal detectors of anti-patterns

Let us assume that some subgraph g, that is part of a software snapshot at revision r, is considered to be an anti-pattern of type t by a respective detector described in Section II. If we have a spatio-temporal classifier for t, then we can find its output for the occurrence of g at revision r according to the method described in the preceding Section V-A. In the proposed approach we will consider g to be an actual anti-pattern of type t iff the output of the spatio-temporal classifier was also t. Conceptually, in this detection strategy we combine a purely static definition of design anti-patterns given in Section II with spatio-temporal knowledge about the evolution of the software. We will consider a graph to be an actual anti-pattern, only when both premises hold.

Clearly, such a compound classifier can reduce the number of false positives but also increase the number of false negatives, thus it does not necessarily improve the quality of an anti-pattern detection. However, in practice, it appears that such a construct improves the classification quality by an average of 4% in terms of F-measure, if we mine the spatio-temporal rules from the very beginning of software evolution and use them to identify static patterns in a separate, final period of this evolution. Details of the experimental setting is given in the following Section VII.

VII. Experimental validation

The experiments were run on the evolution of the following open-source software:

- Argouml ([30], [31], [32]) is a simple UML editor, which used to be popular. The SCM of this software (along with Xerces2j and Jhotdraw) is frequently used as the source of data in mining software repositories research. The analyzed evolution of this software spans from January 1998 to December 2011.
- Struts1 ([33], [34], [35]) is a java web framework, which was popular 20 years ago. The analyzed evolution of this software spans from May 2000 to December 2008.
- Xerces2j ([36], [34], [35]) is a popular Java XML Parser. The analysed software evolution spans from November 1999 to May 2008.
- Elasticsearch ([37], [38]) is a popular search engine. The analyzed evolution of this software spans from February 2010 to September 2017.
- JHotdraw ([39], [40]) is a Java framework for 2D graphics. The analyzed evolution of this software spans from October 2000 to November 2012.
- Lucene-solr ([41], [42], [35]) is a popular search engine. The analyzed evolution of this software spans from September 2001 to November 2016.
- Wildfly ([43], [44], [45]) is a popular Java application server. The analyzed evolution of this software spans from June 2010 to June 2013.

For each system, the spatio-temporal classifier, based on C4.5 Boolean classifier, was trained on the sub-evolution built from the first 70% revisions of the respective system. The closeness and remoteness spatial relations were defined by $d_c = 1$ and $d_r = 2$ respectively (see Section III). The detection quality was tested on each commit of the sub-evolution which consisted of the last 30% of revisions. Table II shows the cases, where the result of detection was different. In two cases the quality decreased by 6-11%, and in all other cases it improved by 1-14% in terms of F1. There was an average improvement of 4%.

VIII. Conclusions

In our previous work ([1], [46]) we analyzed the phenomena of spatio-temporal relations between occurrences of anti-patterns in the software evolution. This paper presents how spatio-temporal rules can help to slightly improve the quality of detection of a few anti-patterns: We combine typical static detectors derived from existing state-of-the-art detection methods with additional knowledge that comes from analysis of the spatio-temporal relations in the software development process.

Dataset	APT	Spatial Prec./Rec.	Spatio-temp. Prec./Rec.	Change of F1
elastic	BB	0.71 / 0.88	1.0 / 0.54	0.89
argouml	Bl	0.9 / 0.76	1.0 / 0.76	1.05
elastic	Bl	0.83 / 0.9	1.0 / 0.88	1.08
Xerces	Bl	0.91 / 0.69	1.0 / 0.69	1.04
elastic	BC	0.87 / 0.84	1.0 / 0.81	1.05
argouml	SAK	0.78 / 1.0	1.0 / 0.94	1.11
elastic	SAK	0.78 / 0.99	1.0 / 0.99	1.14
Lucene	SAK	0.86 / 1.0	1.0 / 0.92	1.04
Xerces	SAK	0.8 / 0.89	1.0 / 0.65	0.94
Xerces	YoYo	0.98 / 1.0	1.0 / 0.99	1.01
			Average	1.04

Table II
IMPACT OF SPATIO-TEMPORAL RULES ON STATIC DETECTION QUALITY. THE TABLE PRESENTS HOW SPATIO-TEMPORAL RULES CHANGE THE QUALITY OF DETECTION OF SPATIAL DETECTORS DESCRIBED IN SUBSECTIONS II-C1–II-C8. THIRD COLUMN PRESENTS PRECISION/RECALL OF PURELY STATIC DETECTION. THE FOURTH COLUMN PRESENTS PRECISION/RECALL AFTER ADDING SPATIO-TEMPORAL RULES. APT = ANTI-PATTERN TYPE, BB = BASE BEAN, BL = BLOB, BC = BRAIN CLASS, SAK = SWISS ARMY KNIFE, F1 = F-MEASURE.

The experimental validation shows that in most cases the prediction quality was identical, with an observable difference only in a few cases described in Table II above. It was worse in only two cases, and on average in improved by 4% in terms of F-measure, which is a harmonic mean of precision and recall.

A. Future work

The following paragraphs provide some proposals for applications and modifications of the proposed framework, which yield to future research in the topic.

In this paper we have presented how spatio-temporal rules can be used to improve the quality of prediction of static pattern detection. The same rules can be used to predict where certain types of anti-patterns may appear in the future in the software source code.

In the method described herein, spatio-temporal rules were trained and used within the same software system. However, it is possible that rules trained on the evolution of one software system can be interpreted within another system. By doing so we may answer if there are universal spatio-temporal rules that model typical spatio-temporal phenomena in the software development process that hold across many projects.

The proposed framework is specifically suited for Java programming language, but can be easily adopted to other programming languages as well. It would require changing the definition of the software snapshot multigraph.

Allens algebra is a helpful formalism, but it can arguably be too simplifying when it is used to model temporal relations between intervals of revisions in the software development process. For example, it cannot measure temporal proximity between intervals. Please note that relation between the separated intervals of revisions are indiscernible in terms of Allens theory in two cases: when they are separated by a single commit and when they are separated by thousands of commits. Thus Allens algebra could be replaced by alternative formalism, which would incorporate more accurate model of temporal relations.

In this research we have assumed time to be linear (i.e. commits are linearly ordered), as we have considered commits from only a single main development branch. In fact, the software development process typically uses many parallel branches and cross-branch merges (see [47]). To cover such phenomena, the time representation in the proposed framework should be replaced with a more versatile model, such as e.g. CTL.

This research is based on the concept of a spatio-temporal relation to occurrences of anti-patterns described in Section II. But it can easily be adapted so that we can use other subgraphs in place of anti-patterns. For example we could use frequent subgraphs ([48]) or graphs built from frequently modified source code entities ([49]).

B. Threats to validity

Drawing general conclusions from empirical studies based on just a few software systems is always difficult, because of the complexity of the matter and the variety of different sources of data. This paper is based on data gathered from systems that share a common characteristic: they are open-source, non-commercial systems, developed by the community for many years. It may appear that software developed differently (e.g. by smaller teams, commercially, with closed source) tends to evolve differently.

Anti-patterns are very infrequent in relation to all possible containment-complete subgraphs in the software, as the number of the latter is exponential in relation to the number of software entities in the source code. To have a balanced set of examples, we have randomly selected sub-set of such subgraphs to construct a decision table described in Section V-A. Even though the results presented in Section VII were stable with multiple repetitions of the experimental reproduction, this fact presumably introduces some randomness in the results.

REFERENCES

[1] Ł. Puławski, "Temporal Relations of Rough Anti-patterns in Software Development," in *Rough Sets*, ser. Lecture Notes in Computer Science, L. Polkowski, Y. Yao, P. Artiemjew, D. Ciucci, D. Liu, D. Ślęzak, and B. Zielosko, Eds. Cham: Springer International Publishing, 2017, pp. 447–464.

[2] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[3] W. H. Brown, R. C. Malveau, H. W. "Skip" McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed. New York, NY, USA: John Wiley and Sons, Inc., 1998.

[4] R. Marinescu, "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws," *Software Maintenance, IEEE International Conference on*, vol. 0, pp. 350–359, 2004.

[5] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," *Empirical Software Engineering and Measurement, International Symposium on*, vol. 0, pp. 390–400, 2009.

[6] R. Wieman, *Anti-Pattern Scanner: An Approach to Detect Anti-Patterns and Design Violations*. LAP LAMBERT Academic Publishing, Nov. 2011.

[7] H. Kagdi, M. L. Collard, and J. I. Maletic, "Towards a taxonomy of approaches for mining of source code repositories," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1–5, May 2005.

[8] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20–36, Jan. 2010.

[9] N. Moha, Y.-g. Gueheneuc, and P. Leduc, "Automatic Generation of Detection Algorithms for Design Defects," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. Tokyo: IEEE, 2006, pp. 297–300.

[10] D. H. Taenzer, M. Ganti, and S. Podar, "Problems in Object-Oriented Software Reuse," in *ECOOP '89: Proceedings of the Third European Conference on Object-Oriented Programming, Nottingham, UK, July 10-14, 1989*, S. Cook, Ed. Cambridge University Press, 1989, pp. 25–38.

[11] A. Stoianov and I. Sora, "Detecting patterns and antipatterns in software using Prolog rules," in *2010 International Joint Conference on Computational Cybernetics and Technical Informatics*. Timisoara: IEEE, May 2010, pp. 253–258.

[12] F. Jaafar, Y. G. Gueheneuc, S. Hamel, and F. Khomh, "Mining the relationship between anti-patterns dependencies and fault-proneness," in *Reverse Engineering (WCRE), 2013 20th Working Conference On*. IEEE, Oct. 2013, pp. 351–360.

[13] T. Zimmermann, "Changes and bugs Mining and predicting development activities," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference On*. IEEE, 2009, pp. 443–446.

[14] C. Izurieta and J. M. Bieman, "A multiple case study of design pattern decay, grime, and rot in evolving software systems," *Software Quality Journal*, pp. 1–35, Feb. 2012.

[15] M. Fowler and K. Beck, *Refactoring Improving the Design of Existing Code*, 1st ed. Addison-Wesley, Jul. 2013.

[16] H. Li and W. Cheung, "An Empirical Study of Software Metrics," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 6, pp. 697–708, Jun. 1987.

[17] T. J. McCabe, "A complexity measure," in *Proceedings of the 2nd International Conference on Software Engineering*, ser. ICSE '76. San Francisco, California, United States: IEEE Computer Society Press, 1976, pp. 407+.

[18] B. A. Nejmeh, "NPATH: A measure of execution path complexity and its applications," *Commun. ACM*, vol. 31, no. 2, pp. 188–200, Feb. 1988.

[19] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[20] M. Hitz and B. Montazeri, "Measuring coupling and cohesion in object-oriented systems," in *Proceedings of International Symposium on Applied Corporate Computing*, 1995, pp. 25–27.

[21] J. M. Bieman and B.-K. Kang, "Cohesion and reuse in an object-oriented system," *SIGSOFT Softw. Eng. Notes*, vol. 20, no. SI, pp. 259–262, Aug. 1995.

[22] N. Moha, Y. G. Gueheneuc, A. F. Le Meur, L. Duchien, and A. Tiberghien, "From a domain analysis to the specification and detection of code and design smells," *Form. Asp. Comput.*, vol. 22, pp. 345–361, May 2010.

[23] M. Fowler, *Patterns of Enterprise Application Architecture*, 1st ed. Addison-Wesley Professional, Nov. 2002.

[24] D. Ratiu, S. Ducasse, T. Girba, and R. Marinescu, "Using History Information to Improve Design Flaws Detection," in *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, ser. CSMR '04. Washington, DC, USA: IEEE Computer Society, 2004.

[25] J. Din, A. B. Al-Badareen, and Y. Y. Jusoh, "Antipatterns detection approaches in Object-Oriented Design: A literature review," in *2012 7th International Conference on Computing and Convergence Technology (ICCCT)*. IEEE, 2012, pp. 926–931.

[26] F. Palomba, R. Oliveto, and A. De Lucia, "Investigating code smell co-occurrences using association rule learning: A replicated study," in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, Feb. 2017, pp. 8–13.

[27] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2014.

[28] M. Zhang, N. Baddoo, P. Wernick, and T. Hall, "Improving the Precision of Fowler's Definitions of Bad Smells," in *2008 32nd Annual IEEE Software Engineering Workshop*. Kassandra, Greece: IEEE, Oct. 2008, pp. 161–166.

[29] J. F. Allen, "Maintaining Knowledge About Temporal Intervals," *Commun. ACM*, vol. 26, no. 11, pp. 832–843, Nov. 1983.

[30] (2015, Sep) Argouml. [Online]. Available: http://argouml.tigris.org/issues/

[31] (2020, Nov) Argouml source code. [Online]. Available: https://github.com/argouml-tigris-org/argouml

[32] (2020, Nov) Argouml source code. [Online]. Available: http://argouml.tigris.org/servlets/ProjectIssues

[33] (2020, Nov) Struts. [Online]. Available: https://struts.apache.org/

[34] (2020, Nov) Apache software foundation scm. [Online]. Available: http://svn.apache.org/repos/

[35] (2020, Nov) Apache software foundation issue tracker. [Online]. Available: https://issues.apache.org

[36] (2020, Nov) Eclipse jdt. [Online]. Available: https://xerces.apache.org/\#xerces2-j

[37] (2020, Nov) Elasticsearch. [Online]. Available: https://www.elastic.co/

[38] (2020, Nov) Elasticsearch source code. [Online]. Available: https://github.com/elastic/elasticsearch

[39] (2018, Apr) Jhotdraw. [Online]. Available: http://www.jhotdraw.org/

[40] (2020, Nov) Jhotdraw source code. [Online]. Available: https://github.com/wrandelshofer/jhotdraw

[41] (2020, Nov) Lucene solr. [Online]. Available: https://solr.apache.org/

[42] (2020, Nov) Lucene solr source code. [Online]. Available: https://gitbox.apache.org/repos/asf/lucene-solr.git

[43] (2020, Nov) Wildfly. [Online]. Available: https://www.wildfly.org/

[44] (2020, Nov) Wildfly scm. [Online]. Available: https://github.com/wildfly/wildfly

[45] (2020, Nov) Wildfly issue tracker. [Online]. Available: https://issues.jboss.org

[46] L. Pulawski, "An automatic approach for detecting early indicators of design anti-patterns," in *JCKBSE*, ser. Frontiers in Artificial Intelligence and Applications, M. Virvou and S. Matsuura, Eds., vol. 240. IOS Press, 2012, pp. 161–170.

[47] V. Driessen. (2010, Jan) https://datasift.github.io/gitflow/IntroducingGitFlow.html. [Online]. Available: https://datasift.github.io/gitflow/IntroducingGitFlow.html

[48] C. C. Aggarwal and H. Wang, *Managing and Mining Graph Data*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[49] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. St. Louis, MO, USA: ACM, 2005, pp. 284–292.