AIBOLIT:
Static Analysis Using Machine Learning

Yegor Bugayenko, Anton Cheshkov, Ekaterina Garmash, Andrey Gusev, Yaroslav Kishchenko, Pavel Lukyanov, Evgeny Maslov, Vitaly Protasov

Huawei Technologies Co., Ltd.
System Programming Lab
Russian Research Institute (RRI)
Moscow, Russia

September 23, 2025

Abstract

Aibolit is a next generation static analyzer powered by machine learning. Aibolit gives recommendations to developers to avoid bad software patterns in order to improve quality of the source code. Aibolit can be extended by adding custom patterns and quality metrics of choice. In this paper, we explain how Aibolit works and how it differs from other static analyzers.

1 Introduction

Insufficient software quality may result in increased development costs and negatively affect customer satisfaction (Jones and Oliver, 2012). *Static code analysis* develops techniques to help detect software quality issues prior to program execution. It has practical applications in various developer tools. There are both open-source[1] (PMD, Rubocop, PHPCS, FindSecBugs, ESLint, Checkstyle, to name a few) and commercial[2] (IBM Security AppScan, PVS-Studio, SonarQube, Parasoft) static analyzers on the market.

Static code analysis can be applied to improve an *internal* and an *external* quality of software (Ilyas and Elkhalifa, 2016). External quality is related to defects encountered by the end user of the software product. Within internal quality, two important subcategories are *functional quality* and *maintainability*. Functional quality is about code correctness and compliance with the functional software specifications (Al Obisat et al., 2018). Code maintainability is about how easy it is to analyze, modify, and adapt given software (Mohammadi et al., 2013).

Functional quality aspects are typically quite susceptible to formal definition and quantification. Functional quality is also an essential requirement in any domain of software development. On the other hand, maintainability is a lot less straightforward to formally specify or quantify. Also, in certain applications it appears less important than functional correctness, although in the business domain it is recognized as an essential property. As a result, there is currently a lot more research and practical tools addressing functional quality aspects of code than maintainability (Gomes et al., 2009). Another aspect of static analysis tools that may have hindered their application to maintainability, is that they are predominantly rule-based. Since there has not yet been a consensus on how to formalize maintainability, it is challenging to devise a set of formal rules to detect it.

We designed our new tool Aibolit to help developers identify patterns in their

[1]PMD: http://pmd.sourceforge.net/, Rubocop: https://github.com/rubocop-hq/rubocop, PHPCS: https://github.com/squizlabs/PHP_CodeSniffer, FindSecBugs: https://find-sec-bugs.github.io/, ESLint: https://eslint.org/, Checkstyle: https://checkstyle.sourceforge.io/.

[2]IBM Security AppScan: https://www.hcltechsw.com/wps/portal/products/appscan, PVS-Studio: https://www.viva64.com/en/pvs-studio/, SonarQube: https://www.sonarqube.org/, Parasoft: https://www.parasoft.com/

code that may cause maintainability issues. It is a next-generation static analysis tool that uses a machine learning (ML) model as an underlying quality prediction mechanism. From the perspective of ML, our product is a recommender system. For a given class file, it gives suggestions to the developer to alter their code. The recommendations come in the form of *code patterns* that are detected in the code and advised to be removed.

Our choice to design Aibolit as an ML-based system alleviates some important shortcomings of rule-based static analyzers. By design, ML algorithms capture statistical relations in the external world (data). Therefore, they can be a good way to model imprecisely and subjectively defined properties of code, such as its maintainability. Moreover, rule-based systems are known to not scale well to the diversity of empirically observed cases, and they tend to get very hard to extend and maintain (**LenatFeigenbaum1987**). The ML approach does not require manual system adaptation as new observations or new features (patterns) come along. In fact, Aibolit provides an easy way for developers to integrate a code pattern of their liking into the recommender system and to analyze the pattern's impact on code quality.

2 Software quality and patterns

2.1 Quality and quality metrics

IEEE Standards define software quality as the array of features of a software product that represent its capability to satisfy specific needs (Boukouchi et al., 2013). Software quality is the extent to which a process, component, or system fulfills customers' needs or expectations through product or service features, thus providing customer satisfaction (Iacob and Constantinescu, 2008).

Functional and structural qualities are the key aspects of software quality (X. F. Liu, Kane, and Bambroo, 2006). Al Obisat et al. (2018) describe functional quality as the capability of the software to properly perform its tasks according to user needs and intended objectives. Structural quality refers to the resilient structure of the code itself and is difficult to test compared to functional quality. The main difficulty is that this notion is quite subjective.

In general, structural code quality is a multi-faceted concept, which covers different attributes of software engineering, for example, maintainability and readability (Mohammadi et al., 2013). To evaluate them, various metrics of software

structure were proposed. For instance, McCabe's software complexity metrics (McCabe, 1976) and cognitive complexity metric (Campbell, 2018a), which are intended to measure readability aspects of the code. Also, for object-oriented systems, a popular set of metrics is the CK suite (Chidamber and Kemerer, 1994). Many approaches apply such metrics suites to distinguish parts of the source code with good or bad quality Filó, Bigonha, and Ferreira (2015), Shatnawi et al. (2010) or to identify code smells (problematic properties and anti-patterns of code) Ouni et al. (2011). However, in general, the software engineering community has not yet reached a consensus as to what exactly structural quality or maintainability is (**Broy2006DemystifyingM**).

2.2 Software patterns and code smells

We understand the term *software patterns* in the most general and abstract way, namely, as any observed code substructures and software solutions. Patterns can be of different scale (from variable and method-level to project level). The term *designed patterns* refers to patterns that are recommended solutions to commonly occurring programming goals and problems (**gamma1995design**). Despite their popularity, there is much controversy about the usefulness and universality of such recommended ways of implementation (**mcconnell2004code**). The software engineering community also identifies patterns that are detrimental to the resulting code. Such patterns are often called *code smells*. These are parts of the source code that contain violations of fundamental design principles and negatively impact maintainability in terms of the ability of the product to evolve, quality of end-product, and developer productivity Reeshti et al. (2019). Din, Al-Badareen, and Jusoh (2012) identified 22 types of code smells in object-oriented design. Kessentini (2019) found a strong correlation between several code smells and software bugs.

For the latter reason, a lot of tools and methods have been designed to avoid code smells. Kreimer (2005) proposes a decision tree-based approach to identify code smells, e.g., long method and large class. Vaucher et al. (2009) apply Bayesian networks to detect God class. Palomba et al. (2015) propose to consider changes of repository history as an input to the code smell detector for computing the list of code components affected by the smell. H. Liu et al. (2019) propose a deep learning based approach to detect code smells.

Code smell detection has been integrated into code inspection tools. Murphy-

Hill and Black (2010) integrate software metrics visualization with a source code view. SonarQube[1] controls and manages the code quality in several ways, such as continuous inspection and issue detection. The platform shows issues like code smells and bugs using lightweight visualizations. Checkstyle[2] and PMD[3] work similarly to SonarQube.

All in all, there is no uniform agreement about which patterns are good and which are bad. We made it our ideology while developing Aibolit: we do not decide what is good or useful *a priori* but let it be inferred from data. By customizing the dataset, quality metric and pattern set, the end user of Aibolit is able to infer which patterns are good for their own end goals.

3 How the Aibolit recommender works

3.1 The Idea

The main purpose of Aibolit is to help developers identify patterns in their code that may cause maintainability issues. From the user perspective, it works by outputting a list of patterns recommended to remove, given a Java class. The Aibolit engine is comprised of two parts: an ML regression model and a recommendation algorithm. The regression model predicts the maintainability of any Java class. The recommendation algorithm uses the regression model to decide which pattern is better to avoid by considering various modifications of the input Java class.

Figure 1 represents the Aibolit recommendation procedure at a high level. In the remaining subsections, we provide a more detailed description of each of Aibolit's components.

3.2 Patterns & Quality metrics

Patterns

As discussed above (Section 2), software engineering researchers and practitioners often associate good and bad code design with specific patterns. We follow

[1] https://www.sonarqube.org/
[2] https://checkstyle.sourceforge.io/
[3] https://pmd.github.io/

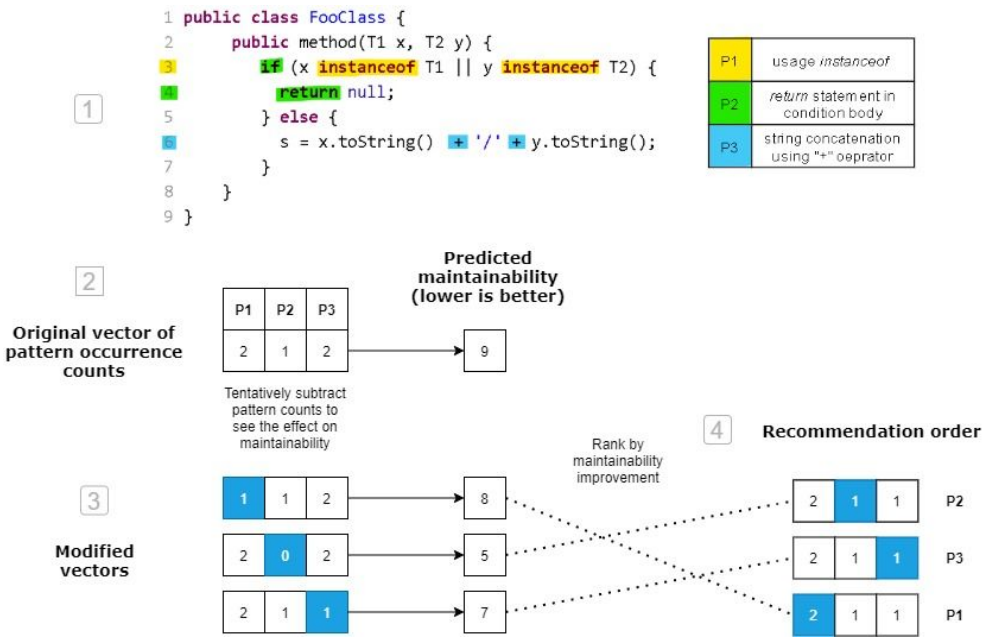

Figure 1: How Aibolit works: (1) Inspect the source code for patterns. (2) Count the pattern occurrences and put them in a vector representation. Compute the maintainability metric value for the vector. (3) Consider changes to the vector representation by subtracting pattern counts. Predict the corresponding maintainability metric score. (4) Rank all the alternative vectors with respect to how much they improve the original maintainability score. Recommend changes that gave the most improvement.

this tradition and build a predictive system to reason about observed patterns in code in terms of their effect on quality. In the current release of Aibolit, the model is built on top of 34 commonly used manually designed patterns as input features. See Appendix for the complete list and detailed descriptions. Note that users of Aibolit can arbitrarily extend the model by implementing and integrating their patterns of choice.

Metrics

The ultimate goal behind Aibolit's approach is to learn to identify maintainability-affecting patterns in code and recommend them to the user. However, as dis-

cussed above, maintainability quantification is still an open problem, and most of the metrics proposed so far typically describe only a narrow aspect of software maintainability. We recognize it as the major challenge of our approach and plan to research this problem in the future.

In the current release of Aibolit, we use Cognitive Complexity (Campbell, 2018b) as the maintainability metric. We refer to it as the maintainability metric or just metric in the remainder of the text.

3.3 Maintainability prediction model

Dataset

To train our prediction model, we mined training data from GitHub open source repositories. We chose repositories written in Java as the main language. We filtered out all non-Java files and all software testing files. To make sure our data is representative of good software engineering standards, we only extracted repositories with at least 100 stars and at least six collaborators.

Aibolit is currently designed to do predictions and recommendations at class level. For simplicity, we only consider files that contain exactly one non-abstract Java class. We filtered out classes with fewer than 50 and more than 300 lines of code. The resulting filtered dataset consists of 124 repositories and 29,065 classes. Before filtering, we split the dataset into test and train sets randomly by files with approximately 0.7:0.3 ratio. The filtered train set contains 20,049 classes, the filtered test set contains 9,016. The complete list of mined repositories and the train/test split is provided in the Aibolit project folder.

Feature and target preprocessing

Each Java class gets associated with a vector of numerical features. We use scaled pattern counts as features. For each pattern from the fixed set (see Appendix), we count its occurrences in the class and divide it by the number of non-commented lines in the class (NCSS). The target value for each datapoint is the maintainability metric value for the class (Section 3.2), also divided by the NCSS. Table 1 gives an illustration of how the dataset looks after this procedure.

We apply NCSS scaling because we observed that our maintainability metric (Cognitive Complexity) is highly correlated with code size. The scaling stimulates the model to find more implicit dependencies between patterns and complexity.

class id	P16	P11	P13	...	CogC (target metric)
class 1	0.008695	0.	0.026086	...	0.417391
class 2	0.	0.05	0.116667	...	0.466667
class 3	0.009909	0.	0.009909	...	0.732673

Table 1: Example of a training dataset with preprocessed feature values. **P16**: *Return null*, **P11**: *Multiple Try*, **P13**: *Null checks*).

Training

We train a gradient boosting regression model (Friedman, 2001). We use the implementation of CatBoost (Dorogush, Ershov, and Gulin, 2018) with the RMSE loss function. For hyperparameter selection, we do a 3-fold cross-validation.

3.4 Recommendation algorithm

Our recommendation algorithm ranks patterns observed in the user's source class according to their individual impact on the code's maintainability metric value. It then outputs a pattern with the *most negative impact* as a recommendation to the user to remove it from their code.

For each pattern p we compute the **impact factor** $I_{neg}(p, C)$ on code C, which is intended to capture the *negative influence* of p on C. It is the difference between the quality metric value of the original code C and the version of C where the count of p has been decreased (Eq. 1):

$$I_{neg}(p_i, C) = M(F(C)) - M(F_{p_i-1}(C)), \tag{1}$$

where M is a quality metric, $F(C)$ is the feature vector $\langle f^C_{p_1}, ..., f^C_{p_n} \rangle$, $F_{p_i-1}(C)$ is the feature vector with the count of p_i decreased by 1.

Under the "lower metric is better" convention (i.e., lower value of quality metric means better quality), lower values of I_{neg} correspond to patterns that contribute more to the deterioration of the code's quality. We rank patterns according to their I_{neg} and output patterns with lowest values as recommendations.

Note that at the moment of recommendation we do not observe code with a decreased pattern count, so we cannot compute the maintainability metric directly. This is why we resort to a predictive maintainability model (Section 3.3), which helps estimate maintainability of a hypothetical code.

Algorithm 1 Aibolit recommendation algorithm

Input: M: pretrained maintainability model; C: class source code; P: array of patterns used for training M

1: $F = []$
2: **for** `i = 1,` $|P|$ **do**
3: $\quad F[i] = \frac{count_{(P[i],C)}}{NCSS_{(C)}}$
4: **end for**
5: $M_{observed} = \mathsf{M}(F)$
6: $I = []$
7: **for** `i = 1,` $|P|$ **do**
8: $\quad F' = F$
9: $\quad F'[i] = F[i] - \frac{1}{NCSS_{(C)}}$
10: $\quad I[i] = M_{observed} - \mathsf{M}(F')$
11: **end for**
12: $I_{worst} = \mathtt{topK}_{i \in [1,\ldots,|P|]}(-I[i])$
13: **return** $\{P[i] \mid i \in I_{worst}\}$

Algorithm 1 summarizes how we do recommendations. For each pattern from the set of patterns used at training, we precompute feature values and compute the metric value of the source code (lines 1-5). Then we compute the impact factor I_p of each pattern p on the source code maintainability (lines 6-10). Under the "lower is better" convention of maintainability metric, low values of I_p indicate that removal of p leads to improvement of the metric score. We collect the K most negatively impacting patterns and output them as recommendations. Thus, we pick patterns p for which I_p are the lowest (lines 12-13).

4 Other usage scenarios

4.1 Empirical analysis of patterns

As a by-product of Aibolit's ML and recommendation engine, we get a tool for empirically analysing different patterns' impacts on the target quality metric. Just like in the main recommendation algorithm (Algorithm 1, Section 3.4), we can estimate whether a particular pattern has a positive or negative impact on the quality metric by considering modifications of source code where pattern count

is decreased or increased. We perform such a procedure on a held-out set, which allows us to estimate the average impact of a particular pattern on quality.

In Table 2 we present a case study of the 34 patterns used at training of Aibolit (see Appendix for the pattern descriptions). On a separate test set (see details in Section 3.3), for each pattern, we considered increasing and decreasing the pattern's count by 1. We used the pretrained Aibolit's regression model to predict the corresponding change in quality metric.

Based on the statistics in Table 2, it appears that *Prohibited class name*, *Count If Return*, *Instance of*, *Null check*, *Nested Loop*, *Array as function argument*, *Joined validation* are **anti-patterns**, since their count decrease tends to improve the metric. Another group of patterns are *Setters*, *Many primary constructors*, *Method chain*, *Non final attribute*, *Super Method*, *Send Null*: for them we observe that decreasing them causes the metric to deteriorate, and increasing causes the metric to improve. We consider them **pro-patterns**. The third group (the rest of the patterns) can both improve and deteriorate the metric. We restrain ourselves from calling them either anti- or pro-patterns.

Attempting to interpret the results, we observe that "true" anti-patterns usually have *if/else condition* or cycle in their definition. E.g., *Null check* always checks for a null, *Count If Return*, *Instance of*, *Joined validation* always have *if condition*, *Nested Loop* always has at least one loop inside. Given that so far we have worked with the Cognitive Complexity metric, it is no surprise that those patterns affect it (*if/else condition* or cycle are the main contributors to *CogC*). Despite this limitation of the present analysis, we believe the proposed *method* itself can be very useful in software engineering practice and research.

4.2 Aibolit Index

In addition to the recommendation functionality, we propose *Aibolit Index*, a score that measures the overall quality of a given software project. It is a single number, with the following properties:

(i) the more patterns are suggested to fix, the higher Aibolit Index;
(ii) the higher negative impact factor (see Section 3.4) of detected patterns, the higher Aibolit Index.

Therefore, the lower Aibolit Index, the better the project's code from the point of view of Aibolit.

Let $P(C)$ be a set of all patterns Aibolit recommends to fix for Java class C.

We define the Aibolit Index $A(C)$ of Java class C as the sum of the products of impact factors I_p and scaled counts $count(p, C)$ for all of the patterns occurring in the class (Eq. 2). We use log-scaling for smoothing purposes, because some patterns are a lot more common than others.

$$A(C) = \sum_{p \in P(C)} I_p(C) \cdot \ln\left(count(p, C) + 1\right) \tag{2}$$

The Aibolit Index of a project is defined as the average Aibolit Index across all Java classes in the project. In Table 3 you can see the calculated Aibolit Index of some GitHub Java repositories with more than 4500 stars. The Aibolit Index is supposed to be a convenient instrument to get a first estimate of the project code's quality.

5 Conclusion & Future Work

Aibolit is a recommender system that helps improve the quality of Java classes. The recommendations are learned from OSS Java projects using ML methods. Aibolit provides ranked recommendations for each specific Java class, which differentiates Aibolit from other style checkers and makes it unique.

Aibolit is an extendable system, allowing anyone to add new patterns and to increase the training dataset, and thus improve the precision and usefulness of recommendations. Aibolit can also be used as a framework for analysis of patterns and to decide whether any pattern, however subjective it is, is an anti-pattern or a pro-pattern with respect to a particular quality metric. As a complementary result, we contribute a 100K+ dataset of patterns and metrics calculated for Java classes.

The first version of Aibolit is relatively simple and there is room for improvement. If an anti-pattern is found, we recommend fixing all instances of the pattern in the code. Instead, we may consider each specific occurrence of the pattern. We may exploit its relative position in the structure of the code, rather than just counting the frequency. Moreover, Aibolit inspects each Java class independently, but we might consider the relations between classes in the future. Furthermore, Aibolit's prediction model relies on patterns only. In order to improve the model, we have to think about additional features, for example, information about the project domain or used frameworks.

Aibolit is a firm step toward the next generation of tools to control and improve software quality. It is a complementary tool for product owners who already use tools to manage software quality.

References

Al Obisat, F. M. et al. (2018). *Review of Literature on Software Quality*. https://www.researchgate.net/.

Boukouchi, Y. et al. (2013). *Comparative Study of Software Quality Models*. https://www.academia.edu/.

Campbell, G. A. (2018a). *COGNITIVE COMPLEXITY: A New Way of Measuring Understandability*.

— (2018b). "Cognitive Complexity: An Overview and Evaluation". In: *Proceedings of the International Conference on Technical Debt*, pp. 57–58. DOI: 10.1145/3194164.3194186.

Chidamber, S. R. and C. F. Kemerer (1994). "A Metrics Suite for Object Oriented Design". In: *IEEE Transactions on Software Engineering* 20.6, pp. 476–493. DOI: 10.1109/32.295895.

Din, J., A. B. Al-Badareen, and Y. Y. Jusoh (2012). *Antipatterns Detection Approaches in Object-Oriented Design: A Literature Review*. https://ieeexplore.ieee.org/.

Dorogush, A. V., V. Ershov, and A. Gulin (2018). *CatBoost: Gradient Boosting With Categorical Features Support*. arXiv: 1810.11363 [cs.LG].

Filó, T. G. S., M. Bigonha, and K. Ferreira (2015). *A Catalogue of Thresholds for Object-Oriented Software Metrics*. https://ufmg.br/.

Friedman, J. H. (2001). "Greedy Function Approximation: A Gradient Boosting Machine". In: *The Annals of Statistics* 29.5, pp. 1189–1232. DOI: 10.1214/aos/1013203451.

Gomes, I. et al. (2009). *An Overview on the Static Code Analysis Approach in Software Development*. https://www.psu.edu/.

Iacob, I. M. and R. Constantinescu (2008). *Testing: First Step Towards Software Quality*. https://www.academia.edu/.

Ilyas, B. and I. Elkhalifa (2016). *Static Code Analysis: A Systematic Literature Review and an Industrial Survey*. https://www.diva-portal.org/.

Jones, C. and B. Oliver (2012). *The Economics of Software Quality*. Addison-Wesley Professional.

Kessentini, M. (2019). *Understanding the Correlation Between Code Smells and Software Bugs*. https://umich.edu/.

Kreimer, J. (2005). "Adaptive Detection of Design Flaws". In: *Electronic Notes in Theoretical Computer Science* 141.4, pp. 117–136. DOI: 10.1016/j.entcs.2005.02.059.

Liu, H. et al. (2019). "Deep Learning Based Code Smell Detection". In: *IEEE Transactions on Software Engineering* 47.9, p. 1. DOI: 10.1109/TSE.2019.2936376.

Liu, X. F., G. Kane, and M. Bambroo (2006). "An Intelligent Early Warning System for Software Quality Improvement and Project Management". In: *Journal of Systems and Software* 79.11, pp. 1552–1564. DOI: 10.1016/j.jss.2005.09.018.

McCabe, T. J. (1976). "A Complexity Measure". In: *IEEE Transactions on Software Engineering* 2.4, pp. 308–320. DOI: 10.1109/TSE.1976.233837.

Mohammadi, N. G. et al. (2013). "An Analysis of Software Quality Attributes and Their Contribution to Trustworthiness". In: *Proceedings of the*

CLOSER, pp. 542–552. DOI: 10.5220/0004502705420552.

Murphy-Hill, E. R. and A. P. Black (2010). "An Interactive Ambient Visualization for Code Smells". In: *Proceedings of the SOFTVIS '10*, pp. 5–14. DOI: 10.1145/1879211.1879216.

Ouni, A. et al. (2011). "Maintainability Defects Detection and Correction: A Multi-Objective Approach". In: *Automated Software Engineering* 20.1, pp. 47–79. DOI: 10.1007/s10515-011-0098-8.

Palomba, F. et al. (2015). "Mining Version Histories for Detecting Code Smells". In: *IEEE Transactions on Software Engineering* 41.5, pp. 462–489. DOI: 10.1109/TSE.2014.2372760.

Reeshti et al. (2019). "Measuring Code Smells and Anti-Patterns". In: *Proceedings of the 4th International Conference on Information Systems and Computer Networks (ISCON)*, pp. 311–314. DOI: 10.1109/iscon47742.2019.9036304.

Shatnawi, R. et al. (2010). "Finding Software Metrics Threshold Values Using ROC Curves". In: *Journal of Software Maintenance and Evolution: Research and Practice* 22.1, pp. 1–16. DOI: 10.1002/smr.404.

Vaucher, S. et al. (2009). "Tracking Design Smells: Lessons From a Study of God Classes". In: *Proceedings of the 16th Working Conference on Reverse Engineering*, pp. 145–154. DOI: 10.1109/WCRE.2009.23.

Appendix

Patterns Dictionary

- **P1. Assert in code**
  Description: If there is an assert statement in a code block, and the name of the class doesn't end with Test, it is considered a pattern.
  Example:

```
class Book {
  void foo(String x) {
  assert x != null; // here
}
```

- **P2. Setter**
  Description: The method's name starts with set, then goes the name of the attribute. There are attributes assigning in the method. Also, asserts are ignored.
  Examples:

```
class Book {
  private String title;
  void setTitle(String t) {
    this.title = t;
  }
}
```

```
class Book {
  private String title;
  public void setIsDiscrete() {
    assert !isDiscrete;
    assert !x; //ignore it
    this.isDiscrete = isDiscrete;
  }
}
```

```
class Book {
private String isDiscrete;

  public void setIsDiscrete(String isDiscretem, boolean x) {
    assert !isDiscrete;
    assert !x; //ignore it
    this.isDiscrete = isDiscrete;
  }
}
```

```
class Book {
```

```
  private String title;

  @Override
  synchronized public void setConf(Configuration conf) {
    this.conf = conf;
    this.randomDevPath = conf.get(
    HADOOP_SECURITY_SECURE_RANDOM_DEVICE_FILE_PATH_KEY,
    HADOOP_SECURITY_SECURE_RANDOM_DEVICE_FILE_PATH_DEFAULT);
    close(); // some minor changes also do not affect, it is
    still Setter pattern
  }
}
```

- **P3. Empty Rethrow**
  Description: We throw the same exception as it was caught.
  Example:

```
class Book {
  void foo() {
    try {
      File.readAllBytes();
    } catch (IOException e) {
      // maybe something else here
      throw e; // here!
    }
  }
}
```

- **P4. ErClass**
  Description: If a class name is one of the following (or ends with this word), it's the pattern:
  Manager, Controller, Router, Dispatcher, Printer, Writer, Reader, Parser, Generator, Renderer, Listener, Producer, Holder, Interceptor.
- **P5. Force type casting**
  Description: The force type casting considered as a pattern.
  Example:

```
// casting to int is
public int square (int n) {
  return (int) java.lang.Math.pow(n,2);
}
```

- **P6. If return if detection**
  Description: If there is a return in if condition, it's a pattern.
  Example:

```
class T1 {
  public void main(int x) {
```

```
    if (x < 0) {
      return;
    } else {
      System.out.println("X is positive or zero");
    }
  }
}
```

- **P7. Implements Multi**
  Description: If a class implements more than 1 interface it's a pattern.
  Examples:

```
public class AnimatableSplitDimensionPathValue implements
    AnimatableValue<PointF, PointF> {
  private final AnimatableFloatValue animatableXDimension;
  private final AnimatableFloatValue animatableYDimension;

  public AnimatableSplitDimensionPathValue(
  AnimatableFloatValue animatableXDimension,
  AnimatableFloatValue animatableYDimension) {
    this.animatableXDimension = animatableXDimension;
    this.animatableYDimension = animatableYDimension;
  }
}
```

```
public class a implements A, B {
}
```

- **P8. Using `instanceof` operator**
  Description: Using the `instanceof` operator is considered a pattern.
  Examples:

```
public static void main(String[] args) {
  Child obj = new Child();
  if (obj instanceof String)
    System.out.println("obj is instance of Child");
}
```

```
class Test
{
  public static void main(String[] args)
  {
    Child cobj = new Child();
    System.out.println(b.getClass().isInstance(c));
  }
}
```

- **P9. Many primary ctors**

Description: If there is more than one primary constructors in a class, it is considered a pattern.
Example:

```
class Book {

  private final int a;
  Book(int x) { // first primary ctor
    this.a = x;
  }
  Book() { // second
    this.a = 0;
  }
}
```

- **P10. Usage of method chaining more than one time**
  Description: If we use more than one method chaining invocation.
  Example:

```
// here we use method chaining 4 times
public void start() {
  MyObject.Start()
  .SpecifySomeParameter()
  .SpecifySomeOtherParameter()
  .Execute();
}
```

- **P11. Multiple Try**
  Description: Once we see more than one try in a single method, it's a pattern.
  Example:

```
class Foo {
  void bar() {
    try {
      // some code
    } catch (IOException ex) {
      // do something
    }
      // some other code
    try {  // here!
      // some code
    } catch (IOException ex) {
      // do something
    }
  }
}
```

- **P12. Non final attributes**
  Description: Once we see a mutable attribute (without final modifier), it's considered a pattern.
  Example:

```
class Book {
  private int id;
  // something else
}
```

- **P13. Null checks**
  Description: If we check that something equals (or not equals) null (except in constructor) it is considered a pattern.
  Example:

```
class Foo {
  private String z;
  void x() {
    if (this.z == null) { // here!
      throw new RuntimeException("oops");
    }
  }
}
```

- **P14. Partial synchronized**
  Description: Here, the synchronized block doesn't include all statements of the method. Something stays out of the block.
  Example:

```
class Book {
  private int a;
  void foo() {
    synchronized (this.a) {
      this.a = 2;
    }
    this.a = 1; // here!
  }
}
```

- **P15. Redundant catch**
  Description: Here, the method `foo()` throws IOException, but we catch it inside the method.
  Example:

```
class Book {
  void foo() throws IOException {
    try {
      Files.readAllBytes();
```

```
    } catch (IOException e) { // here
      // do something
    }
  }
}
```

- **P16. Return null**
  Description: When we return null, it's a pattern.
  Example:

```
class Book {
  String foo() {
    return null;
  }
}
```

- **P17. String concatenation using + operator**
  Description: Any usage of string concatenation using the + operator is considered a pattern match.
  Example:

```
public void start() {
  // this line matches the pattern
  System.out.println("test" + str1 + "34234" + str2);
  list = new ArrayList<>();
  for (int i = 0; i < 10; i++)
    list.add(Boolean.FALSE);
}
```

- **P18. Override method calls parent method**
  Description: If we call a parent method from an overridden class method, it is considered as the pattern.
  Example:

```
@Override
public void method1() {
  System.out.println("subclass method1");
  super.method1();
}
```

- **P19. Class constructor except `this` contains other code**
  Description: The first constructor has this() and some other statements. This is the "hybrid constructor" pattern.
  Example:

```
class Book {
  private int id;
  Book() {
    this(1);
```

```
    int a = 1; // here
  }
  Book(int i) {
    this.id = i;
  }
}
```

- **P20_5, P20_7, P20_11. Line distance between variable declaration and first usage greater than threshold**
  Description: If the line distance between variable declaration and first usage exceeds some threshold, we consider it as the pattern. We calculate only non-empty lines. P20_5 means that the distance is 5.
  Example:

```
// variable a declared and used with 2 lines distance
static void myMethod() {
  String path1 = "/tmp/test1";
  int a = 4;

  String path2 = "/tmp/test2";
  String path3 = "/tmp/test3";
  a = a + 4;
}
```

- **P21. Variable is declared in the middle of the method body**
  Description: All variables we need have to be declared at the beginning of their scope. If a variable is declared inside the scope following after logical blocks, we consider that this is the pattern.
  Example:

```
// The declaration of variable list matches the pattern.
static void myMethod2() {
  int b = 4;
  b = b + 6;
  List<Integer> list = new List<Integer>();
}
```

- **P22. Array as argument**
  Description: If we pass an array as an argument, it's a pattern. It's better to use objects, instead of arrays.
  Example:

```
class Foo {
  void bar(int[] x) {
  }
}
```

- **P23. Joined Validation**

Description: Once you see a validation (if with a single throw inside) and its condition contains more than one condition joined with OR – it's a pattern.
Example:

```
class Book {
  void print(int x, int y) {
    if (x == 1 || y == 1) { // here!
      throw new Exception("Oops");
    }
  }
}
```

- **P24. Class declaration must always be `final`**
  Description: Once you see a non `final` class, it's a pattern.
  Example:

```
class Book {
  private static void foo() {
  }
}
```

- **P25. Private static method**
  Description: Once you see a `private static` method, it's a pattern.
  Example:

```
class Book {
  private static void foo() {
    //something
  }
}
```

- **P26. Public static method**
  Description: Once you see a `public static` method, it's a pattern.
  Example:

```
class Book {
  private static void foo() {
    //something
  }
}
```

- **P27. Var siblings**
  Description: Here fileSize and fileDate are "siblings" because they both have file as first part of their compound names. It's better to rename them to size and date.
  file and fileSize are NOT siblings.
  Example:

```
class Foo {
  void bar() {
    int fileSize = 10;
    Date fileDate = new Date();
  }
}
```

- **P28. Assign null**
  Description: Once we see `= null`, it's a pattern.
  Example:

```
class Foo {
  void bar() {
    String a = null; // here
  }
}
```

- **P29. Multiple `while` pattern**
  Description: Once you see two or more `while` statements in a method body, it's a pattern.
  Example:

```
class Book {
  void foo() {
    while (true) {
    }
    // something
    while (true) {
    }
  }
}
```

- **P30. Protected method**
  Description: Once we find a protected method in a class, it's a pattern.
- **P31. Send `null`**
  Description: Once we see that `null` is being given as an argument to some method, it's a pattern.
  Example:

```
class Foo {
  void bar() {
    FileUtils.doIt(null); // here
  }
}
```

- **P32. Nested loop**
  Description: Once we find a loop (`for` / `while`) inside another loop it's a pattern.

Example:

```
class Foo {
  void foo() {
    while (true) {
      for (;;) { // here
      }
    }
  }
}
```

patterns	p- m-	p+ m+	p- m+	p+ m-	p- m=	p+ m=
Asserts	92	61	186	219	5	3
Setters	245	160	901	976	21	31
Empty Rethrow	77	65	25	36	0	1
Prohibited class name	617	311	218	522	0	2
Force Type Casting	2363	2313	1742	1790	14	16
Count If Return	969	883	214	298	0	2
Implements Multi	459	320	264	403	0	0
Instance of	1396	1374	151	173	6	6
Many primary constructors	19	19	604	605	2	1
Method chain	573	574	2217	2214	43	45
Multiple try	371	239	259	391	0	0
Non final attribute	1249	1185	5835	5839	127	187
Null check	5863	5978	575	457	9	12
Partial synchronized	46	49	155	149	0	3
Redundant catch	84	46	38	79	4	1
Return null	1290	813	926	1401	4	6
String concat	1596	2089	1506	1012	23	24
Super Method	212	275	1012	949	10	10
This in constructor	15	42	72	45	0	0
Var declaration distance for 5 lines	1976	1464	738	1244	19	25
Var declaration distance for 7 lines	1190	959	733	949	16	31
Var declaration distance for 11 lines	703	603	365	466	10	9
Var in the middle	3372	3117	2799	3046	37	45
Array as function argument	882	768	351	464	1	2
Joined validation	232	226	14	26	8	2
Non final class	7141	3201	3723	7663	0	0
Private static method	529	541	667	648	2	9
Public static method	1166	1212	1142	1088	4	12
Null Assignment	1245	803	717	1150	6	15
Multiple While	120	97	16	39	0	0
Protected Method	868	402	1147	1613	7	7
Send Null	556	325	1510	1733	5	13
Nested Loop	580	527	29	81	1	2

Table 2: Empirical analysis of patterns. For each source code in the test set, we consider increasing (**p+**) and decreasing (**p-**) pattern count by 1. We recorded whether the maintainability metric increased (**m+**) or decreased (**m-**) as a result of that (where lower is better). In some cases the metric did not change (**m=**). The values in the cells are counts of cases.

Repository	Aibolit Index	Total files	Total NCSS	GitHub stars
ReactiveX\RxJava	6.66	1493	25270	42972
bumptech\glide	5.81	465	5078	29364
JakeWharton\butterknife	6.31	74	935	25347
greenrobot\EventBus	7.48	51	651	22637
skylot\jadx	6.69	602	12658	22602
alibaba\fastjson	9.97	144	20175	21891
alibaba\druid	6.36	822	28852	21581
Netflix\Hystrix	6.74	292	2920	19888
ReactiveX\RxAndroid	5.47	9	59	19010
google\gson	6.87	160	2941	18084
square\picasso	6.21	26	687	17514
libgdx\libgdx	5.02	1981	46409	17105
nostra13\Android-Universal-Image-Loader	7.84	62	1059	16722
qiurunze123\miaosha	3.57	197	841	16224
wuyouzhuguli\SpringAll	13.41	467	463	15221
justauth\JustAuth	4.36	46	241	8916
heibaiying\BigData-Notes	11.17	54	204	7166
crossoverJie\cim	6.70	96	574	5841
wildfirechat\server	6.51	285	4671	5140
febsteam\FEBS-Shiro	5.66	90	453	4777

Table 3: Aibolit Index of some popular Java repositories.