UDC 004.02 **DOI: 10.20998/2079-0023.2020.01.19**

I. Y. MALIK, V. Y. VOLOVSHCHYKOV, V. F. SHAPO, M. A. GRINCHENKO

TECHNOLOGY OF IDENTIFYING ANTIPATTERNS IN ANDROID PROJECTS WRITTEN IN KOTLIN LANGUAGE

The problem of the lack of instruments for identifying the characteristics of low-quality code in Android projects that are written in the Kotlin language is determined. A review of modern approaches for identifying antipatterns in program code is accomplished. The analysis of the methods used to find problems with code in Android projects is performed. DECOR and Paprika approaches are considered. Conclusions are drawn about the importance of finding design flaws in program code for the mobile software development and its further support. An antipatterns identification approach for Kotlin language program code in Android projects is proposed. An algorithm for identifying low-quality Kotlin code is presented. The technology for detecting poor quality code characteristics consists of four stages: collecting metrics about an analyzed software system, building a quality model, converting a quality model into a graph representation, and identifying predefined antipatterns. The collection of metrics, including the search for both Android-specific and object-oriented metrics of Chidamber and Kamerer, is proposed to be implemented through parsing source code and converting it into an abstract syntax tree using the KASTree library. The implementation of KASTree library usage is offered through the Adapter design pattern. The construction of a quality model is implemented using the Paprika tool, supplemented by a number of introduced metrics. Conversion of quality model exactly into graph representation is used to identify antipatterns in order to ensure the speed and quality of complex queries execution for identifying antipatterns. Antipatterns identification using database queries is based on various template rules, including the Catolino rules. Different features of applying the Cypher query language to a graph database are used to represent the rules in form of queries. Results of the work can be used in development of software for poor quality code identification in mobile applications written in Kotlin language, as well as in studies of mobile development antipatterns for this language.

Keywords: antipattern, identification, graph model, low-quality code, Kotlin, Android, Adapter pattern

І. Ю. МАЛІК, В. Ю. ВОЛОВЩИКОВ, В Ф. ШАПО, М. А. ГРИНЧЕНКО

ТЕХНОЛОГІЯ ІДЕНТИФІКАЦІЇ АНТИПАТЕРНІВ В КОДІ ANDROID ПРОЕКТІВ МОВОЮ KOTLIN

Визначена проблема відсутності програмного забезпечення для ідентифікації характеристик низькоякісного коду в проектах Android, що розроблені з використанням мови програмування Kotlin. Проведено огляд сучасних підходів до виявлення анти-шаблонів в програмному коді. Виконано аналіз методів, що використовуються для знаходження проблем з кодом для Android проектів. Розглянуто відомі підходи до ідентифікації: DECOR та Paprika. Зроблено висновки про важливість знаходження недоліків у програмному коді для розробки мобільного програмного забезпечення та його майбутнього обслуговування. Запропоновано підхід до ідентифікації антипатернів у програмному коді Kotlin для Android проектів. Представлено алгоритм ідентифікації неякісного коду. Технологія визначення характеристик неякісного коду включає чотири етапи: збір метрик про програмну систему, що аналізується, побудова моделі якості, конвертація моделі якості в графове представлення та ідентифікація наперед визначених антипатернів. Збір метрик, що включає пошук як Android-специфічних, так і об'єктно-орієнтованих метрик Чидамбера та Камерера, пропонується реалізувати через синтаксичний аналіз вихідного коду та його конвертацію в абстрактне синтаксичне дерево з використанням бібліотеки KASTree. Впровадження бібліотеки KASTree пропонується через шаблон проектування Адаптер. Побудова моделі якості реалізується засобами інструменту Paprika, що доповнено низкою введених метрик. З метою забезпечення швидкості та якості виконання складних запитів для ідентифікації антипатернів використовується конвертація моделі якості саме в графове представлення. Безпосередньо ідентифікація антишаблонів за допомогою запитів використовує в якості основи різноманітні шаблонні правила, у тому числі правила Католіно. Для представлення правил у вигляді запитів використовуються можливості застосування мови запитів Cypher до графової бази даних. Результати роботи можуть бути використані при розробці програмного забезпечення для ідентифікації неякісного коду в мобільних застосунках, що написані мовою Kotlin, а також при дослідженні антипатернів в мобільній розробці з використанням даної мови.

Ключові слова: антипатерн, ідентифікація, графова модель, неякісний код, Kotlin, Android, патерн Адаптер

И. Ю. МАЛИК, В. Ю. ВОЛОВЩИКОВ, В. Ф. ШАПО, М. А. ГРИНЧЕНКО

ТЕХНОЛОГИЯ ИДЕНТИФИКАЦИИ АНТИПАТТЕРНОВ В КОДЕ ANDROID ПРОЕКТОВ НА ЯЗЫКЕ KOTLIN

Определена проблема отсутствия средств идентификации характеристик некачественного кода в проектах Android, которые написанные на языке Kotlin. Проведен обзор современных подходов к выявлению анти-шаблонов в программном коде. Выполнен анализ методов, используемых для нахождения проблем с кодом в Android проектах. Рассмотрены известные подходы по обнаружению антипаттернов в коде: DECOR и Paprika. Сделаны выводы о важности нахождения недостатков в программном коде для разработки мобильного программного обеспечения и его будущей поддержки. Предложен подход к идентификации антипатернов в программном коде Kotlin для Android-проектов. Представлен алгоритм идентификации некачественного кода. Технология определения характеристик некачественного кода включает четыре этапа: сбор метрик об анализируемой программной системе, построение модели качества, конвертация модели качества в графовое представление и идентификация заранее определенных антипаттернов. Сбор метрик, включающий поиск как Android-специфических, так и объектно-ориентированных метрик Чидамбера и Камерера, предлагается реализовывать через синтаксический анализ исходного кода и конвертацию его в абстрактное синтаксическое дерево с использованием библиотеки KASTree. Внедрение библиотеки KASTree предлагается через шаблон проектирования Адаптер. Построение модели качества реализуется средствами инструмента Paprika, дополненного рядом введенных метрик. С целью обеспечения скорости и качества выполнения сложных запросов для идентификации антипаттернов используется конвертация модели качества именно в графовое представление. Непосредственно идентификация антипаттернов с помощью запросов использует в качестве основы различные шаблонные правила, в том числе правила Католино. Для представления правил в виде запросов используются возможности применения языка Cypher к графовой базе данных. Результаты работы могут быть использованы при разработке программного обеспечения для идентификации некачественного кода в мобильных приложениях, написанных языком Kotlin, а также при исследованиях антипатернов в мобильной разработке с использованием указанного языка.

Ключевые слова: антипаттерн, идентификация, графовая модель, некачественный код, Kotlin, Android, паттерн Адаптер

Introduction. The field of information systems development has existed for a long time. There are currently projects that have been supported for over 10 years and systems are being developed that will be maintained for a long time to come. In such circumstances, it is important to maintain the quality of the software product consistently. One of the main characteristics of quality software (SW) is the quality of the source code. Its readability, ease of understanding, simplicity of support and refactoring play an important role in supporting the project [1]. Therefore, writing quality code is an important task of software engineering (SE).

Poor quality code characteristics are programming patterns that indicate potential source code issues [2]. These characteristics are different from software errors because they do not affect the correctness of the application. However, these patterns provoke the creation of a system that is more difficult to develop and maintain, which can make the software more vulnerable to errors.

At present, most studies related to the characteristics of low-quality code focus on desktop and web-based applications. However, nowadays, mobile platforms are becoming more popular [3]. Therefore, it is an important task to investigate the characteristics of poor quality code specific to mobile development. It follows that one of the most important tasks is to develop a detector that can identify the flaws in the source code of mobile projects.

Analysis of literature and major achievements. With the advent of mobile applications as new software systems, many authors have begun to study mobile-specific code problems. Reimann [4] proposed a directory of 30 high-quality factors of poor quality code for Android. These factors cover various aspects such as software implementation, user interface, and database usage. After these, characteristics were identified and described by Reimann. Scientists began to propose tools and approaches for their detection [5, 6, 7].

Much of the research on code defects in Android applications focuses on examining the effect of basic factors of problematic code [2]. For example, Linarez-Vasquez [8] used the DECOR tool [9] to detect object-oriented antipatterns in mobile applications developed with J2ME. DECOR, in turn, is based on peer inspections, DSL, and automatic generation of identification algorithms.

Rasool [10] examined the existence of traditional poor-quality code criteria [2] in Android applications written in Java to determine whether they are more common in the mainstream classes. Mainstream classes are classes in the Android project that are inherited from Android SDK classes, such as Activity, Fragments, Services. The author states that the basic classes tend to suffer from God class, Long method and Switch operator because of their character, since they perform most of the functionality of the software. The author also found that a Long list of methods is less likely to appear in basic classes because their methods are inherited from classes defined in the Android SDK.

Hecht [5] developed a tool to detect poor-quality Paprika code for 8 features for the Java programming language. He created his own approach using a graph model. The author searched for defects in 15 popular Android applications, including Facebook, Skype and Twitter. The author claims that traditional code smells are as common in Android as non-Android applications.

Mannan [11] conducted a large-scale empirical study to compare the prevalence and impact of antipatterns on mobile and desktop applications. The author found that while the density of code problems is the same in both mobile and desktop systems, some of them are more common in mobile applications. For example, Data class is more common in mobile applications, while duplication of code is more inherent in desktop systems.

Mohammed Ilyas Azeem in his work [12] analyzed machine learning techniques that were investigated to identify low-quality code. He concluded that most existing studies use decision trees or the support vector method as machine learning algorithms. However, the problem of creating the optimal configuration has not been properly solved.

Problems of identifying low-quality code in Android projects. Analysis [5–9] found a large number of studies devoted only to the Java programming language. Java remains one of the most popular programming languages in the world and for Android projects in particular. However, nowadays, Kotlin programming language is also becoming popular [13], which is being developed rapidly and is supported by many developers all over the world. Kotlin is included in the list of officially supported languages for developing Android applications. Since May 7, 2019 it is the recommended language for Android application development. However, no tools were found that could detect low-quality code written on Kotlin. There is still a problem with the lack of methods, technologies or algorithms for detecting poor quality code characteristics for projects created using Kotlin. The main cause of this problem can be considered the relative youth of the language. The stable version was released only 4 years ago. Research into identifying code issues for Kotlin is very important.

Another problem is the presence of non-identifiable characteristics. Since the beginning of antipatterns studies and their classification, there have been various attempts to identify the characteristics available. However, a review [5, 6, 8, 10] confirmed that developed applications can only detect some of the code issues described. Fig. 1 shows how often different antipatterns have been studied in the literature. From fig. 1 it follows that some characteristics are studied more frequently, while there are still poor quality code factors for which no relevant studies have been conducted. Of all the characteristics of the bad quality code identified by Fowler [2], there are still those that are not determined by any identification technique. There are more than thirteen different techniques in the literature regarding 21 code defects, and 16 methods have been developed for the God class only. However, none of them shows the above mentioned defects. It can be concluded that not all code flaws are currently identifiable.

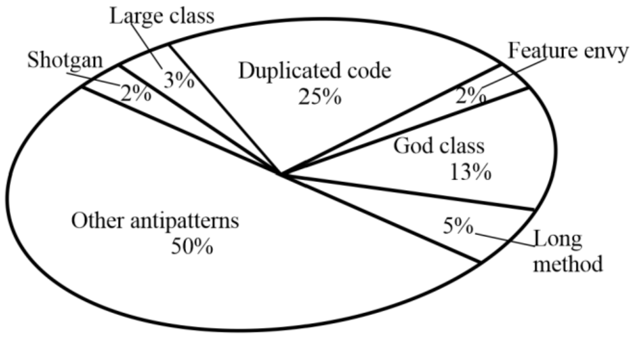

Fig. 1 Study frequency of individual characteristics

Analysis of existing methods. The literature [5, 9] describes two main approaches to identifying the characteristics of low-quality code in Android applications. The first one is used in software called DECOR. Its main idea is to use the expertise knowledge to build the classification and taxonomy that generates the identification algorithms. Another method was developed for Paprika SP. It is based on the metrics of the application being analyzed and the construction of quality and graph models. Based on the latter, source code defects are detected.

Fig. 2 illustrates a common algorithm for identifying antipatterns in code. A key element is the identification approach, which varies by tool. However, it receives input on the characteristics of poor-quality code and software metrics generated from source or compiled code.

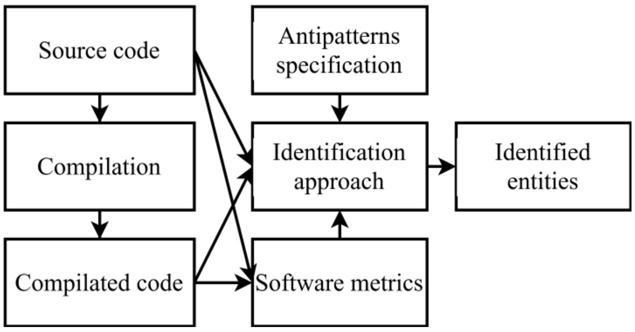

Fig. 2 General identification algorithm

The antipattern detection technology in DECOR uses a four-step algorithm. At the first step, the experts analyze the subject domain and identify the key concepts on the basis of which the classification and taxonomy of all characteristics of poor quality code are made. The second step is the specification of factors using domain-specific language (DSL). The key concepts are formalized in the form of rule cards, where a card is a set of rules which contains characteristics that describe a particular antipattern. DSL allows to determine the links, properties, and internal structure of the antipattern using metrics. The rule cards then automatically generate an algorithms for issue identifying with code using DSL and the source code parser. In the last step, the generated algorithms are automatically applied to system models and identify suspicious classes.

DECOR is designed with features of Java language, its syntax and corresponding code-writing convention. Therefore, in the study [9], all metrics are calculated according to this programming language. In addition, the vocabulary and taxonomy were developed for only four antipatterns. This means that DECOR can only identify them. The disadvantage of this approach is a great dependence on peer inspections. The first two steps of the algorithm are not automated, and adding a new antipattern for identification will be time-consuming and will require experts.

The approach used in the Paprika tool contains a three-step algorithm. The first step is to collect metrics. Input is one APK files and related metadata. Output is a Paprika quality model that includes entities, metrics, and properties. At this step generates a mobile app model and removes quality metrics from the input artifact. Paprika builds the model based on 6 entities. 17 properties describe entities and attach to them as attributes. Properties and entities are united by connections. Paprika also pulls metrics for each entity. There are currently 34 metrics available. The method uses 2 types of metrics: object-oriented and Android-specific. Unlike properties, metrics require the calculation or processing of byte-code. The quality model is built using the described parameters. The second step is the conversion of the quality model into a graph model. The input is a model received at the previous step. The output is a graph model stored in the database. Because graph databases are independent of the rigid scheme, the graph model is almost the same as the first step model. All entities are represented as vertices of a graph. Attributes and metrics are properties of vertices. The connections between the entities are represented by unidirectional edges. The last step is the identification of antipatterns. Input is a graph database containing a quality model. Output is vertices, and therefore entities containing antipatterns. Once the model is downloaded and indexed by the graphical database, you can use the database query language to identify common characteristics of poor-quality code.

Because Paprika analyzes byte-code, this means that this tool can only analyze Java-written applications. In addition, the byte-code often fails to get accurate metric estimates. This was stated by the author himself in his research [5].

Formulation of the problem. An analysis of works [5–10] designates two major problems with identifying poor-quality code in Android projects: the lack of methods and tools for Kotlin and the presence of unexplored programming antipatterns. Kotlin [13] has been identified by Google as being a recommended development tool for Android, which is rapidly developing and gaining popularity. Considering also that there are only four poor-quality code unexplored factors, it can be concluded that the first problem is more critical. In addition, it should be noted that more than 8% [13] of all developers use Kotlin all the time. Thus, it can be said that the topic of research of the identification of problem code for Android projects written in Kotlin is relevant.

According to the analysis [5–9], two main methods for identifying poor-quality code were determined. Because the DECOR approach is not fully automated, the authors will not rely on it for research. The approach used in Paprika is more promising, as the author revealed not only general characteristics, but also Android specific ones. However, the results of the metric calculation, and therefore the identifications will be more accurate when analyzing the source code. It will also expand the list of used metrics. This approach will increase the number of identifiable antipatterns. Therefore, it is proposed in the future to improve the Paprika technique by analyzing the source code and to use it to identify Kotlin code flaws.

Thus, the purpose of the work is to investigate and improve the method of identifying poor-quality code for Android projects written in Kotlin.

Low-quality code identification technology. Identification technology developed by the authors builds on four-step approach. Generalized scheme of it is shown in fig. 3. First step includes syntax analysis of project source code. In contrast to Paprika [5], where byte code is analyzed, it was decided to work with source code. This has the following advantages: elimination if information loss, higher accuracy of obtained results, ready-made source code metrics can be used, original names of components are preserved. In addition, byte code is stored in archives, which imposes additional restrictions on operation with system. Source data can be both a link to a project directory and a link to a web hosting service where project is stored (GitHub, Gitlab, Bitbucket etc.). It is suggested to use the KASTree library for syntax analysis. It allows to present the source code as an abstract syntax tree (AST). On the next stage quality model is constructed based on the obtained AST. The syntax tree provides information about classes, methods, variables and relationships between them. Unlike the Paprika quality model this approach also includes object-oriented Chidamber and Kemerer metrics [14]. In the third step after building quality model, it is converted for saving into graph DB. On the last stage identification is performed by calling prepared queries to DB. Queries are needed for searching antipatterns. After that, a report is created listing found code flaws and their location. We briefly describe content of each of the steps in the next sections.

Syntax analysis is a process of converting source code into structured representation. It is needed for building an AST, which can help quickly get needed metrics for generating the quality model. AST is a tree representation of the abstract syntactic structure of source code written in a programming language.

AST is a tree data structure which is a finite set T with the following properties:

- There is only one root T of the tree – project directory;
- Other nodes T_i are syntactic constructions found in the source code;
- All non-root nodes are distributed among disjoint sets and each set is a subtree;

$$T = \bigcup_{i=1}^{m} T_i\,,\ \bigcap_{i=1}^{m} T_i = \emptyset,$$

where m – number of syntax constructions.

AST can be obtained from a Kotlin project using KASTree library. However, the result of this tool is a syntax constructions list that are not a unified data format.

It is suggested to use Adapter design pattern to provide flexibility and extensibility of the system, which is shown in fig. 4. It converts KASTree library output to a common JSON data format. In case of changing syntax parsing instrument it is not needed to change logic of using AST on the next stage.

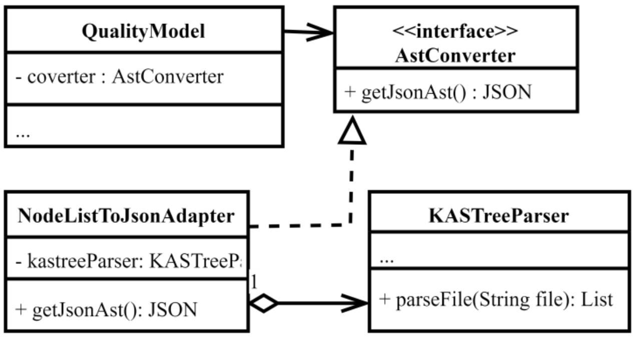
QualityModel
- coverter : AstConverter
...
<<interface>> AstConverter
+ getJsonAst() : JSON
NodeListToJsonAdapter
- kastreeParser: KASTreeP
+ getJsonAst(): JSON
1
KASTreeParser
...
+ parseFile(String file): List

Fig. 4. Adapter design pattern for converting AST

Quality model generation. This model is based on the quality model which used in Paprika instrument [5]. It includes 6 entities: Package, Class, Method, Attribute, Variable and Argument. Each entity is described by attributes, such as full name, access modifier, type and others. The model provides 7 types of relationships between entities: Package Has Class, Class Has Method, Class Has Attribute, Method Has Argument, Inherits, Calls, Uses. Relationships exist between two determined types of entities. For example, relation Inherits can exists only between two entities of type Class. In addition to attributes entities has source code metrics. Model provides 34 metrics. They are dived into object-oriented (OO) and Android-specific metrics. OO metrics consists of simple

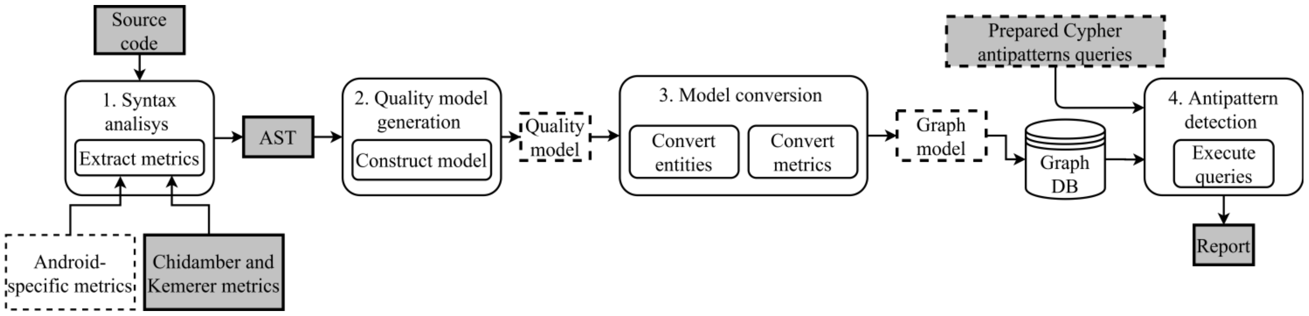

Fig. 3. Low-quality code identification technology for Android projects written in Kotlin

and computational values. Simple measures can be obtained directly from the AST, e. g. number of methods in a class, number of parameters in a method and so on. On the other hand, computing metrics requires additional calculations.

Authors propose to supplement this data with Chidamber and Kemerer metrics. Their usage expands the range of code flaws that can be identified and improves the accuracy of the results. We briefly describe these metrics.

Weighted Methods per Class (WMC). Consider class C with set of methods $m_1,m_2,\dots m_n$, which are defined in this class. Let $c_1,c_2,\dots c_n$ cyclomatic complexity of methods. Then:

$$WMC=\sum_{i=1}^{n} c_i\,.$$

Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and the directed edge connects two nodes if the second command might be executed immediately after the first command. Then:

$$c_i=E_i-N_i+2, i\in\overline{1,n},$$

where E_i – number of edges in graph for i-th method;
N_i – number of nodes in graph for i-th method.

Depth of Inheritance (DIT). This metric is used to determine the location of a class in the inheritance hierarchy. DIT shows how many class ancestors can potentially affect the class. DIT is defined as a maximum number of ancestral classes per class. It is needed recursively bypass the inheritance tree before reaching the first ancestor class to find this metric. The number of attended classes is DIT.

Coupling Between Objects (CBO) for a class is a count of the number of other classes to which it is coupled. Coupling between two classes is said to occur when one class uses methods or variables of another class. COB is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends. Let class C with set of methods $m_1,m_2,\dots m_n$ and set of variables $p_1,p_2,\dots p_k$, which are used in this class. Herewith $m_i\notin C$, $p_j\notin C$, $i=\overline{1,n}$, $j=\overline{1,k}$. Then:

$$CBO=\sum_{i=1}^{n} m_i+\sum_{j=1}^{k} p_j\,.$$

Response for a Class (RFC) is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class. This includes all methods accessible within the class hierarchy. RFC is defined as follows:

$$RFC=\{M\}\bigcup_{i}\{R_i\},$$

where $\{R_i\}$ – set of methods called by i–th method;
$\{M\}$ – set of methods, which belong to class.

Lack of Cohesion in Method (LCOM) measures the extent to which methods reference the classes instance data. Consider a class C with set of methods $m_1,m_2,\dots m_n$. Let $\{I_j\}$ is set of instance variables used by method m_i. There are n such sets $\{I_1,I_2,\dots,I_n\}$. Let $P=\{(I_i,I_j|I_i\cap I_j=\emptyset)\}$, and $Q=\{(I_i,I_j|I_i\cap I_j\neq\emptyset)\}$. If all n sets are empty then let P also is empty. Then:

$$LCOM=\begin{cases}|P|-|Q|, |P|>|Q|,\\ \quad 0, |P|\le|Q|.\end{cases}$$

Catolino described [15] rules for determining code flaws using described metrics. As shown below, it is possible to identify code smells by converting these rules into database queries.

Transformation into a graph representation. Model must be presented as a graph for convenient and efficient operation of it. If entities of the quality model are considered as vertices of the graph, relationships between entities as edges, and attributes and entity metrics as properties of vertices, then the quality model can be converted to graph form. This graph is stored in memory using a graph DB. This solution is flexible and efficient, because such approach of data storage does not depend on a rigid scheme. Thereby converted model (fig. 4) is the same as that described in previous section. In addition, graph repositories show high performance with datasets up to 2^{35} nodes and relationships. This allows to identify antipatterns even on large systems.

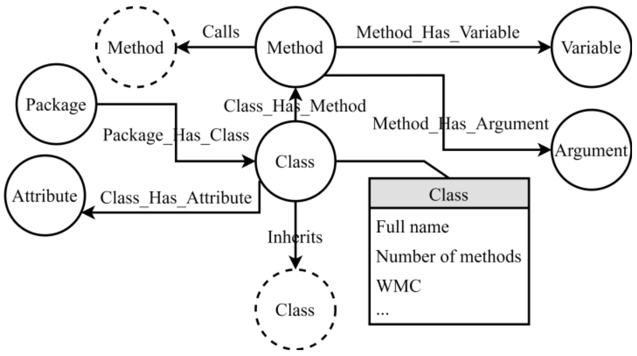

Fig. 4. Schematic representation of a graph model

Antipatterns search. It is suggested to identify code smells based on the model stored in DB. Information about the structural parts of the code which implement antipatterns can be obtained by querying graph DB. A report is built based on this information. Let show searching code flaws on two widespread antipatterns: God Class as object-oriented and Internal getters and setters as Android specific.

God class is a class that contains a large number of fields and methods. It is responsible for different logic, its attributes are related to different processes, which implies strong connection with other classes. Such classes are difficult to maintain and increase the complexity of software modification. Author [15] proposes to use metrics such as WMC, LCOM, number of methods (NM) and number of fields (NF) for identification God Class cases. If for any class C LCOM >15 and WMC >9 or NM >12 and NF >8, then it is considered as God Class. The graph

DB query in Cypher notation for identification such antipatterns is shown in fig. 5.

```
MATCH (c1:Class)
WHERE
    c1.lack_of_cohesion_in_methods > 15
    AND c1.weighted_methods_per_class > 9
    OR c1.number_of_methods > 12
    AND c1.number_of_attributes > 8
RETURN c1
```

Fig. 5. Cypher query to search for God class

In Android, class fields should be available directly for performance reasons. Usage of internal getters and setters turns into a virtual call, making the operation three times slower than direct access. Internal getters and setters can be identified using the graph model. Query for this antipattern is shown in fig. 6. This query looks for two methods from one class, when one calls the other, designated as a getter or setter.

```
MATCH (m1:Method)-[:CALLS]->(m2:Method),
     (c1:Class)
WHERE
     (m2.is_setter OR m2.is_getter)
     AND c1-[:CLASS_OWNS_METHOD]->m1
     AND c1-[:CLASS_OWNS_METHOD]->m2
RETURN m1
```

Fig. 6. Cypher query to identify Internal getters and setters

Conclusions. This work describes a technology of identifying poor-quality code for Android projects written in Kotlin. It is based on the work of Hecht [5] and is an option to improve the Paprika tool and adapt it to the Kotlin programming language. The proposed approach uses source code instead of byte-code and complements the object-oriented metrics offered by Hecht. This will increase the number of antipatterns of identification and, using the work [14, 15], improve the accuracy of the results. The implementation of the described technology will effectively identify both object-oriented and Android-specific characteristics of poor quality code. As a future extension of the study, authors suggest to use proposed approach in developing software of antipatterns identification in Kotlin web-based applications or adapt it for Swift language, which is used in developing projects for iOS platform.

References

1. Counsell S., Rob M. H., Hamza H., Black S. Exploring the eradication of code smells: An empirical and theoretical perspective. *Advances in Software Engineering*. 2010, vol. 2010, p. 12. doi:10.1155/2010/820103.
2. Fowler M. Refactoring: *Improving the Design of Existing Code*. Boston: Addison-Wesley Professional, 2018. 448 p.
3. Rashedul I., Rofiqul I., Tahidul Arafhin M. Mobile application and its global impact. *International Journal of Engineering and Technology*. 2010, vol. 10, iss. 6, pp. 72–78.
4. Reimann J., Brylski M. A tool-supported quality smell catalogue for Android developers. *Softwaretechnik-Trends*. 2015, vol. 34, no. 2, pp. 44-46.
5. Hecht G., Rouvoy R., Moha N., Duchien L. Detecting antipatterns in Android apps. Lille: INRIA, 2015. 24 p.
6. Kessentini M., Ouni A. Detecting Android smells using multi-objective genetic programming. *ICMSES*. 2017, pp. 122–132. doi:10.1109/MOBILESoft.2017.29.
7. Palomba F., Di Nucci D., Panichella A. Lightweight detection of Android-specific code smells: the aDoctor project. *ICSAER*. 2017, 12 p. doi:10.1109/SANER.2017.7884659.
8. Linarez-Vasquez M., Klock S., McMillan C. Domain matters: bringing further evidence of the relationships among antipatterns, application domains, and quality-related metrics in Java mobile apps. *ICPC*. 2014, pp. 232–243. doi: 10.1145/2597008.2597144.
9. Moha N., Duchien L., Gueheneuc Y. DECOR: a method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*. 2010, vol. 36, pp. 20–36. doi: 10.1109/TSE.2009.50.
10. Rasool G., Ali Arab A. Recovering Android Bad Smells from Android Applications. *Springer Berlin Heidelberg*. 2020, pp. 1–27. doi: 10.1007/s13369-020-04365-1.
11. Mannan A. M., Ahmed I., Almurshed R. A. M. Understanding code smells in Android applications. *ICMSES*. 2016, pp. 225–236. doi: 10.1109/MobileSoft.2016.048.
12. Azeem M.I., Palomba F., Shi, L., Wang, Q. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information & Software Technology*. 2019, vol. 108, pp. 115–138.
13 Kotlin 2019 the state of Developer Ecosystem in 2019 Infographic. URL: https://www.jetbrains.com/lp/devecosystem-2019/kotlin/ (дата звернення 04.02.2020).
14 Chidamber S. R., Kemerer C. F. A metric suite for object oriented design. *IEEE Transactions on Software Engineering.* 1994, vol. 20, pp. 476–493.
15 Catolino G. Improving change prediction models with code smell-related information. Empir Software. 2019, p. 42.

References (transliterated)

1. Counsell S., Rob M. H., Hamza H., Black S. Exploring the eradication of code smells: An empirical and theoretical perspective. *Advances in Software Engineering*. 2010, vol. 2010, p. 12. doi:10.1155/2010/820103.
2. Fowler M. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley Professional, 2018. 448 p.
3. Rashedul I., Rofiqul I., Tahidul Arafhin M. Mobile application and its global impact. *International Journal of Engineering and Technology*. 2010, vol. 10, iss. 6, pp. 72–78.
4. Reimann J., Brylski M. A tool-supported quality smell catalogue for Android developers. *Softwaretechnik-Trends*. 2015, vol. 34, no. 2, pp. 44-46.
5. Hecht G., Rouvoy R., Moha N., Duchien L. Detecting antipatterns in Android apps. Lille: INRIA, 2015. 24 p.
6. Kessentini M., Ouni A. Detecting Android smells using multi-objective genetic programming. *ICMSES*. 2017, pp. 122–132. doi:10.1109/MOBILESoft.2017.29.
7. Palomba F., Di Nucci D., Panichella A. Lightweight detection of Android-specific code smells: the aDoctor project. *ICSAER*. 2017, 12 p. doi:10.1109/SANER.2017.7884659.
8. Linarez-Vasquez M., Klock S., McMillan C. Domain matters: bringing further evidence of the relationships among antipatterns, application domains, and quality-related metrics in Java mobile apps. *ICPC*. 2014, pp. 232–243. doi: 10.1145/2597008.2597144.
9. Moha N., Duchien L., Gueheneuc Y. DECOR: a method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*. 2010, vol. 36, pp. 20–36. doi: 10.1109/TSE.2009.50.
10. Rasool G., Ali Arab A. Recovering Android Bad Smells from Android Applications. *Springer Berlin Heidelberg*. 2020, pp. 1–27. doi: 10.1007/s13369-020-04365-1.
11. Mannan A. M., Ahmed I., Almurshed R. A. M. Understanding code smells in Android applications. *ICMSES*. 2016, pp. 225–236. doi: 10.1109/MobileSoft.2016.048.
12. Azeem M.I., Palomba F., Shi, L., Wang, Q. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information & Software Technology*. 2019, vol. 108, pp. 115–138.
13 Kotlin 2019 the state of Developer Ecosystem in 2019 Infographic. Available at: https://www.jetbrains.com/lp/devecosystem-2019/kotlin/ (accessed 04.02.2020).

14 Chidamber S. R., Kemerer C. F. A metric suite for object oriented design. *IEEE Transactions on Software Engineering*. 1994, vol. 20, pp. 476–493.

15 Catolino G. Improving change prediction models with code smell-related information. *Empir Software*. 2019, p. 42.

Received 22.04.2020

Відомості про авторів / Сведения об авторах / About the Authors

Малік Іван Юрійович – бакалавр, Національний технічний університет «Харківський політехнічний інститут», студент; м. Харків, Україна; ORCID: https://orcid.org/0000-0003-1015-0603; e-mail: malik.ivan.yurich@gmail.com

Воловщиков Валерій Юрійович – кандидат технічних наук, доцент, Національний технічний університет «Харківський політехнічний інститут», доцент кафедри програмної інженерії та інформаційних технологій управління; м. Харків, Україна; ORCID: https://orcid.org/0000-0003-4454-2314; e-mail: valera@kpi.kharkov.ua

Шапо Владлен Феліксович – кандидат технічних наук, доцент, Національний університет «Одеська морська академія», доцент кафедри автоматичного управління і обчислювальної техніки; м. Одеса, Україна; ORCID: https://orcid.org/0000-0002-3921-4159; e-mail: stani@te.net.ua

Гринченко Марина Анатоліївна – кандидат технічних наук, доцент, Національний технічний університет «Харківський політехнічний інститут», професор кафедри стратегічного управління; м. Харків, Україна; ORCID: https://orcid.org/0000-0002-8383-2675; e-mail: marinagrunchenko@gmail.com

Малик Иван Юрьевич – бакалавр, Национальный технический университет «Харьковский политехнический институт», студент; г. Харьков, Украина; ORCID: https://orcid.org/0000-0003-1015-0603; e-mail: malik.ivan.yurich@gmail.com

Воловщиков Валерий Юрьевич – кандидат технических наук, доцент, Национальный технический университет «Харьковский политехнический институт», доцент кафедры программной инженерии и информационных технологий управления; г. Харьков, Украина; ORCID: https://orcid.org/0000-0003-4454-2314; e-mail: valera@kpi.kharkov.ua

Шапо Владлен Феликсович – кандидат технических наук, доцент, Национальный университет «Одесская морская академия», доцент кафедры теории автоматического управления и вычислительной техники; г. Одесса, Украина; ORCID: https://orcid.org/0000-0002-3921-4159; e-mail: stani@te.net.ua

Гринченко Марина Анатольевна – кандидат технических наук, доцент, Национальный технический университет «Харьковский политехнический институт», профессор кафедры стратегического управления; г. Харьков, Украина; ORCID: https://orcid.org/0000-0002-8383-2675; e-mail: marinagrunchenko@gmail.com

Malik Ivan Yuriyovich – bachelor, National Technical University "Kharkiv Polytechnic Institute", student; Kharkiv, Ukraine; ORCID: https://orcid.org/0000-0003-1015-0603; e-mail: malik.ivan.yurich@gmail.com

Volovshchykov Valeriy Yuriyovich – Candidate of Technical Sciences, Docent, National Technical University "Kharkiv Polytechnic Institute", Associate Professor of the Department of Software Engineering and Management Information Technologies; Kharkiv, Ukraine; ORCID: https://orcid.org/0000-0003-4454-2314; e-mail: valera@kpi.kharkov.ua

Shapo Vladlen Felixovitch – Candidate of Technical Sciences, Docent, National University "Odessa Maritime Academy", Associate Professor of the Theory of Automatic Control and Computing Machinery Department; Odessa, Ukraine; ORCID: https://orcid.org/0000-0002-3921-4159; e-mail: stani@te.net.ua

Grinchenko Marina Anatoliyvna – Candidate of Technical Sciences, Associate Professor, National Technical University "Kharkiv Polytechnic Institute", Professor of the Department of Strategic Management; Kharkiv, Ukraine; ORCID: https://orcid.org/0000-0002-8383-2675; e-mail: marinagrunchenko@gmail.com