Code Based Selected Object-Oriented Mechanisms Identification

Kristián Jablonický[1], Ján Lang[1]

[1]Institute of Informatics, Information Systems and Software Engineering, Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Ilkovičova 2, 842 16 Bratislava 4, Slovakia

Abstract

This article deals with select mechanisms that are provided by the object-oriented programming language Java. Using these mechanisms is crucial for a proper approach to object-oriented programming. However, this is often difficult for many programmers that are new to writing code in the said language. This paper looks at these mechanisms, defines them by quoting various sources, and proposes a solution that spots these described mechanisms in code. It does so by reading source code from .java files inputted by its user, gathering information about the defined and used classes, interfaces, methods, etc. With this information gained it outputs data which would often require a larger context of the code and good knowledge of the files' contents. Thanks to the strict definitions and rules, that will be stated later in this paper, the application can consistently spot these mechanisms in code and inform its user about the fact. As a result, the application can help programmers that are new to object-oriented programming languages by ensuring them, that they implemented these mechanisms correctly, or by informing them about their absence. The core mechanisms described in this paper are inheritance, encapsulation, class association, and polymorphism.

Keywords

programming, Java, object-oriented, refactoring, inheritance, encapsulation, class association, polymorphism

1. Introduction

Object-oriented mechanisms are widely used in object-oriented programming languages such as C#, C++, Python, and Java to approach coding from a different perspective. These mechanisms allow programmers to write code with a different paradigm that is more easily upgradable, enables code reusability, data hiding, and so on. Many people that first start writing code in such programming languages have a background in procedural programming languages such as C, Fortran, and Pascal. Since these languages do not support the implementation of most object-oriented mechanisms, programmers have to learn how to implement these new mechanisms wisely from scratch. Definitions of these object-oriented mechanisms that can be found online, in books, or in articles are often vague, and thus their implementations in code can be varied. This is why this article dives into these mechanisms and gives them strict and precise definitions. The solution proposed spots these select mechanisms in code and informs

SQAMIA 2023: Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, September 10–13, 2023, Bratislava, Slovakia

kristianjablonicky@gmail.com (K. Jablonický); jan.lang@stuba.sk (J. Lang)

0009-0008-4124-9482 (K. Jablonický); 0000-0002-3271-7271 (J. Lang)

the user about which mechanisms were (or were not) detected. The solution itself does not hint to the user what changes should be made to the code or how to interpret the data - that is left up to the user. Doing so could help with obtaining a deeper understanding of these mechanisms, their usage, and their impact on the code. The rest of the paper is organized as follows, Section 2, which follows up this introduction, shows related works which provide similar tools to the one proposed in this article, or sources dealing with similar topics. Section 3 states how the proposed solution functions and what features it provides its users with. On top of that, it quotes various definitions of mentioned mechanisms from other sources and states the application's strict definitions. Section 4 demonstrates how the proposed solution works when put to use on various source codes, either written for testing purposes or source codes of open-source projects. Lastly, section 5 wraps up what was discussed and discovered in this article.

2. Related work

Projects and applications with similar goals to ours have been made in the past, but the differences in their implementation make our solution more suitable for some uses. A number of books were written that take real-life examples and use them as inspiration for programmers to write better object-oriented code. These books include [1], [2], [3], and [4], to name a few. These books provide great explanations for many of the mechanisms this paper discusses, which is why they will be cited a number of times in the following section. Eclipse Metrics plug-in is an IDE extension of Eclipse that provides the user with various object-oriented mechanism usage statistics[1]. As its title suggests, it focuses on providing various metrics, rather than on pointing out where said mechanisms are detected. This tool will be used later in the fourth chapter. There are many tools that claim to detect design patterns from source code. The article [5] proposes such a tool, however, the tool can no longer be installed on computers with newer operating systems. The website from which the tool can be downloaded lists links to related works, but none of these links currently work.

Some tools such as the ones proposed in [6] and [7] implement bots that automatically provide their users with suggestions on how to improve their code. These tools are however fairly hard to set up and are mainly focused on aiding the user with improving a source code, not on understanding and analyzing it.

There are so many articles published that deal with this topic, that a number of articles were published that compare these solutions. Articles [8], [9], and [10], to name a few, compare the features and accuracies of these tools. The tool proposed in [11] shows great potential and will be brought up again in the evaluation section.

Refactoring is another field that deals with similar tasks. The published book [12] states, that refactoring represents a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. The article [13] states a similar definition, and adds that Refactoring is the process of changing the structure of software without changing its behavior. It is widely practiced by developers, and considerable research and development effort has been invested in refactoring tools. However, the fairly recent article [6] informs that empirical studies have shown that these solutions are not widely

[1]Frank Sauer. Eclipse Metrics plugin. Last visited at (2nd of January 2023): https://metrics.sourceforge.net/

accepted by software developers and most refactorings are still performed manually. The paper agrees that continuous refactoring is necessary to maintain source code quality and to cope with technical debt. It also adds that manual refactoring is inefficient and error-prone, which is why these automated refactoring solutions have been proposed in the past. To mention a few of these works, the article [14] proposes a tool, which allows its users to refactor their code with features such as "rename method", "rename field", "add method argument" etc. What is also important is that this article allows its readers to take a closer look at how exactly this process works. Similarly, [7] cites the previous work as an influence, and proposes an Eclipse IDE plugin, which suggests its users changes that aim to improve their code without altering its function. Since the tool proposed in this paper is a stand-alone application and does not rely on any refactoring library or IDE APIs, it has to work carefully with the source code. The following section explains what steps it has to take to provide the user with correct information.

3. Object-Oriented Mechanisms Identification Solution Proposal

This section briefly goes over the principles upon which the proposed solution is built. This includes prerequisites, steps it takes during its runtime, and outputs that the whole process leads to. The solution is an application written in Python code that takes the user's specified folder path and searches for all Java source code files within that folder, as well as folders nested within these folders. The files are open in succession, and their contents are read line by line. These lines get altered so that the program can rely on certain things that may not necessarily be true for the raw source code. Once all source code is read, altered, and sorted, the program goes over all the lines with now larger context about the source code to detect mechanisms that were impossible to detect before. Once all this is done, all the detected information gets communicated to the program's user.

It is assumed that the inputted files' content is fully valid and is contained in ".java" or ".aj" type files. An inputted file that was not compiled before may contain uncompilable code, and thus our solution might give the user incorrect or irrelevant information.

The before-mentioned alterations to the read source code include: the removal of all string values; the removal of all comments (both single-line and multi-line); the removal of symbols that imply a variable is an array or a list when the information is irrelevant to our application; making each keyword and symbol separated by exactly one space character (this means either squeezing multiple white characters or adding exactly one); the removal of all ";" symbols

The following insights regarding how certain mechanisms are detected will make it clearer why such alterations are made.

3.1. Class hierarchies

As [1] states, inheritance is the most important “is a” hierarchy, and is an essential element of object-oriented systems. Basically, inheritance defines a relationship among classes, wherein one class shares the structure or behavior defined in one or more classes (denoting single inheritance and multiple inheritances, respectively). Inheritance thus represents a hierarchy of abstractions, in which a subclass inherits from one or more superclasses. Typically, a subclass

augments or redefines the existing structure and behavior of its superclasses. Hierarchy is defined as a ranking or ordering of abstractions.

Inheritance in code is the easiest to detect, since it is declared by the use of the keyword " extends ". The application can rely on there being exactly one space character before and after the keyword, because of the steps described in the third section. Most other object-oriented mechanisms unlike this one can not be detected by simply searching for a string, which makes those mechanisms harder to spot in code. A class can only extend a single other class (but can be extended by multiple). Once the solution detects the said keyword, it inserts the current class' declaration that extends a different class into an inheritance tree as a new node. Since all the keywords and class/interface names are separated with exactly one space character, the solution can calculate the index at which class names start and end.

Once all the source code is read and the tree gets expanded, the application prints out to the user which class extends which, and at which line in the source code this happens. Then it prints out all the inheritance trees formed in the program's runtime.

3.2. Encapsulation

Encapsulation is one of the core elements of Object-oriented programming. As [1] puts it, encapsulation is most often achieved through information hiding (not just data hiding), which is the process of hiding all the secrets of an object that do not contribute to its essential characteristics; typically, the structure of an object is hidden, as well as the implementation of its methods.

As the application goes through the separated contents of all the class' bodies it searches for hints of encapsulation. When a line containing a declaration of an attribute is read, the application checks whether this attribute has a "private" or "protected" access modifier. If not, this attribute is globally accessible by all the other classes and thus not encapsulated properly. This lack of encapsulation is communicated to the application's user. On the other hand, when all the lines of a class are read, and no non-encapsulated attributes were spotted, the class is considered to be encapsulated as a whole. Furthermore, the program points out all the attributes that are encapsulated correctly - an encapsulated attribute has at least one "setter" or "getter" method defined. This serves as an encouragement for the user to approach attributes this way.

3.3. Class association

Class association is another crucial Object-oriented mechanism. This includes aggregation and composition, both will be discussed respectively since their detection in code is vastly different.

As [2] states, general class association is especially useful, when we first start looking into relationships between classes, since at that point we do not know what exactly are they used for. Further, [3] states that it is the simplest and most general kind of relationships, which simply indicates that the objects of the two classes are related in some non-hierarchical way. An association implies that an object of one class is making use of an object of another class and is indicated simply by a solid line connecting the two class icons.

Aggregation is a special type of association, according to [1] it is a "part of" type hierarchies that describe aggregation relationships. Aggregation permits the physical grouping of logically

related structures, and inheritance allows these common groups to be easily reused among different abstractions.

A less direct kind of aggregation is also possible, called composition, which is containment by reference. The textbook [3] says that composition implies that each instance of the part belongs to only one instance of the whole and that the part cannot exist except as part of the whole.

Association is the simplest one to detect. The program states that two classes are associated when there is a relation between them, but it is not aggregation nor composition. Relations between classes are detected by finding a mention of another class in a class' body. There are two things the program needs to look out for in order to spot any type of association. A method of another class is being called, or an object of another class' type is being declared and/or used.

Composition is a special type of association, where the class' constructor creates a new object of another class' type. This means that this class' objects contain other class' object/objects as attributes.

There are three different ways to detect aggregation: a class' method returns an object of another class' type; a class' constructor takes another class' object as a parameter; a class' attribute is of another class' type, and composition was not detected.

3.4. Polymorphism

According to [1], polymorphism is perhaps the most powerful feature of object-oriented programming languages next to their support for abstraction, and it is what distinguishes object-oriented programming from more traditional programming with abstract data types. Polymorphism represents a concept in type theory in which a single name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass. Any object denoted by this name is therefore able to respond to some common set of operations.

The following criteria have to be met in order for the application to detect polymorphism: an object is declared of a certain class type; the previously declared object is upcast by being instantiated to another class; the upcast object calls a method that is overridden from the superclass.

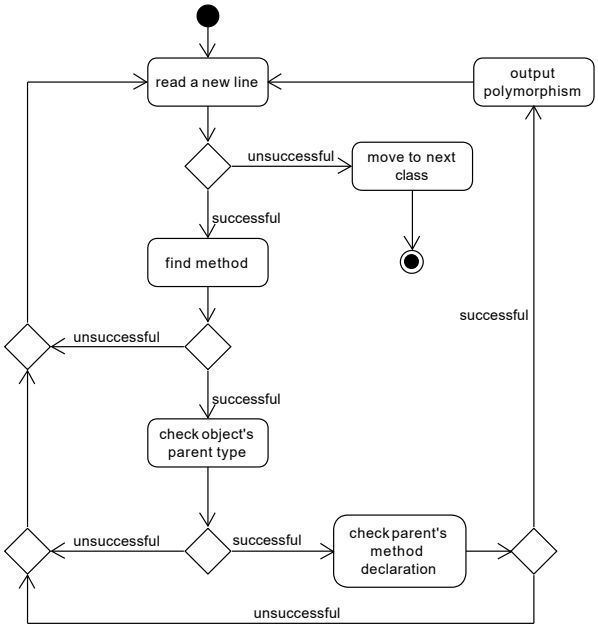

Figure 1: UML diagram demonstrating the detection of polymorphism

The figure 1 depicts a UML diagram that shows how this process works.

To describe the process a bit more precisely, first, the solution finds a mention of another

class' name on a line. Then it makes sure that an object of said class is being declared. Then it searches for the object's upcast, which could look like the following:

```
classA objName;
objName = new classB ();
```

This example shows a "classA" type object called "objName", which is shortly upcast to "classB" type.

Lastly the application searches for an instance of this upcast object calling a method. If it succeeds, it makes sure that this method is overridden (the object's type superclass declares a method with the same name).

Another way to detect an upcast object is when it is received as a method parameter. When an object of a type that is extended by at least one other class is found as a method argument, it is safe to assume that the parameter could receive an upcast object.

The last way of detecting upcast objects is searching for object declaration in a "for" cycle expression. An object is declared in a "for" loop only if the loop iterates over a list of objects. The list may contain both objects of a class or objects of a class that extends the mentioned class in the "for" loop. Thus the application assumes all the objects within this list to be potentially upcast.

Now that all the core object-oriented mechanisms are introduced, defined, and their detection in code is specified, the following section can evaluate the results such detection methods lead to.

Listing 1: Source code example utilizing many object-oriented mechanisms

```
interface I1 {
    public void method();
}
interface I2 {
    public int get_i();
}
class C1 implements I1, I2 {
    private int i;
    private C3 obj;
    C1(){
        obj = new C3();
        i = 0;
    }
    public int get_i(){
        return i;
    }
    public void method(){
        System.out.println("C1 method");
    }
}
class C2 extends C1 implements I1, I2{
    int i;
    C3 obj;

    C2(C3 obj){
        obj = obj;
    }

    public int get_i(){
        return i;
    }
    public void method(){
        System.out.println("C2 method");
    }
}
class C3 implements I1 {
    public void method(){
        System.out.println("C3 method");
    }
}
class example {
    public static void main(String[] args)
        {
        C1 obj = new C1();
        C3 obj_2 = new C3();
        obj = new C2(obj_2);
        obj.method();
    }
}
```

4. Evaluation

The following example demonstrates many of the features mentioned in the previous section. It is a dummy Java source code file that was inputted to the proposed application [2].

The content of the source code can be seen in the listing 1.

This gives results in the following output shown in figure 2.

```
-Hierarchies:
---Inheritance hierarchy 1---
C1
└── C2

---Interface hierarchy 1---
I1
├── C1
├── C2
└── C3

---Interface hierarchy 2---
I2
├── C1
└── C2
```

```
-Encapsulations:
"C1" class encapsulates the "obj" attribute.
"C1" class encapsulates the "i" attribute.
The "C2" class doesn't encapsulate
all of its attributes!

-Associations:
The "C1" type composites the "C3" type.
The "C2" type aggregates the "C3" type.
The "example" type is associated with the "C1" type.
The "example" type is associated with the "C3" type.
The "example" type is associated with the "C2" type.

-Polymorphisms:
Class example:
 The "obj" object implements polymorphism.
```

Figure 2: Detailed text output of the proposed application

The output at first shows which lines from the source code implement an interface or extend a class. This is then followed by a visual representation of what kind of hierarchies those lines of code lead to. In this example, there is a single inheritance hierarchy and two interface hierarchies. Since a class can implement multiple interfaces, classes C1 and C2 can be seen in both interface hierarchies.

The next information outputted is that the C1 class encapsulates the "obj" and "i" attributes. That is because both of these attributes have private access modifiers. The reason why these lines were printed however is that the "i" attribute has a getter method defined.

The next outputted line says that the C1 class composites the C3 class. This is because a C3 type object is defined in the C1 class' constructor. The "outside world" does not have access to this object, only the objects of C1 class type do.

The next line states that the C2 class aggregates the C3 class. This is similar to the previous line, the C2 class' constructor sets the value of a C3 type object attribute. This time the difference is that the C3 type object gets received as a parameter of the constructor. This means that we know that the "outside world" has access to this very object, which is what sets aggregation and composition apart.

[2]The Python script's source code, as well as the links to its compiled executable files and the user's manual were last visited at (24th of August 2023): https://github.com/KristianJablonicky/OopIdentification

After that, the output mentions that the C2 class does not encapsulate all of its attributes. This is because the integer "i" attribute does not have an access modifier.

The following lines state that the example class is associated with the C1, C2, and C3 classes since objects of such types are defined in this class' "main" method.

Lastly, the output states that the "obj" object implements polymorphism. This is because at first, its type is C1, but two lines later in the source code it gets upcast to the C2 class type. Once it is upcast, it calls the "method" method, which is overridden from the C1 superclass, thus this object implements polymorphism.

The following example will not have its source code shown due to its size, but it is available for download here[3]. The example will be used to demonstrate the output on a larger scale project and have its output compared with the output provided by similar tools. Selected part of the output can be seen in 3.

```
-Hierarchies:
---Nested type hierarchy 1---
ATM
├── authenticate
├── Addcheck
├── Deletecheck
├── balancecheck
├── Depositcheck
├── Withdrawcheck
├── Exitcheck
├── Backcheck
├── BCheck
├── BClear
├── Nextcheck
└── Prevcheck
-Encapsulations:
"Account" class encapsulates the
"username" attribute.
The "ATM" class doesn't encapsulate all
of its attributes!
-Associations:
"ATM" type composites the "Screen" type.
"ATM" type composites the "Keypad" type.
"ATM" type composites the "CashDispenser" type.
"ATM" type composites the "DepositSlot" type.
"ATM" type composites the "BankDatabase" type.
"ATM" type aggregates the "Iterator" type.
"ATM" type is associated with the "authenticate" type.
"ATM" type is associated with the "Addcheck" type.
"ATM" type is associated with the "Deletecheck" type.
"ATM" type is associated with the "balancecheck" type.
"ATM" type is associated with the "Depositcheck" type.
"ATM" type is associated with the "Withdrawcheck" type.
"ATM" type is associated with the "Exitcheck" type.
"ATM" type is associated with the "Account" type.
"ATM" type is associated with the "Transaction" type.
"ATM" type is associated with the "Backcheck" type.
"ATM" type is associated with the "BalanceInquiry" type.
"ATM" type is associated with the "Withdrawal" type.
"ATM" type is associated with the "Deposit" type.
"ATM" type is associated with the "Nextcheck" type.
"ATM" type is associated with the "Prevcheck" type.
"ATM" type is associated with the "BCheck" type.
"ATM" type is associated with the "BClear" type.
-Polymorphisms:
Class BankDatabase:
 The "currentAccount" object implements polymorphism.
-Design patterns:
ATM's method getinstance implements
the design pattern Singleton.
```

Figure 3: Chosen outputs of the application provided after inputting the ATM project's source code

Note that this paper has not explained class nesting nor Singleton design principle, but their detection is implemented and thus was outputted by the program.

First, hierarchies will be discussed. The Class Visualizer tool[4] serves as a quick UML class diagram generator, but it can also be used to evaluate class association and hierarchies. Figure 4

[3]Link to the ATM-Machine project's GitHub page is(7th of April): https://github.com/DanH957/ATM-Machine

[4]Class Visualizer, made by Jonatan Kaźmierczak, last visited at (7th of May 2023): http://www.class-visualizer.net/index.html

shows all the relations classes that formed with the "ATM" class that the tool was able to detect.

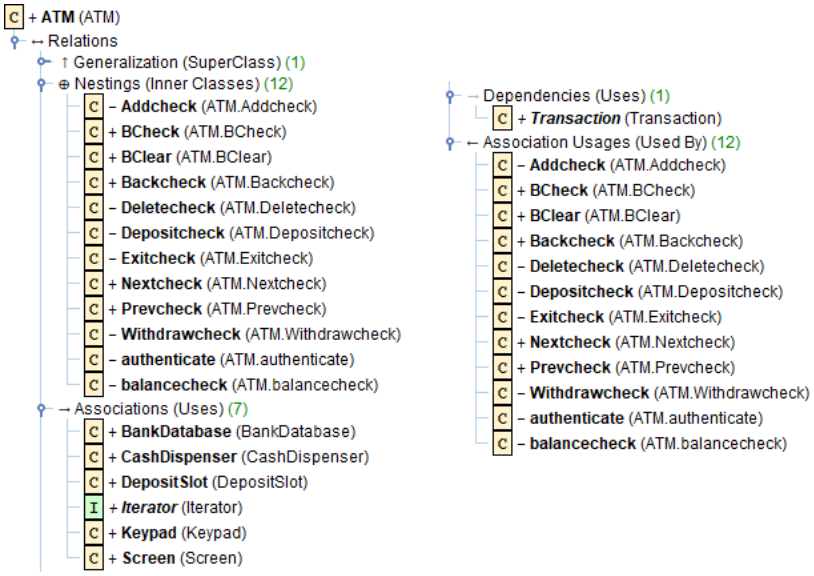

Figure 4: List of all the relations including the ATM class that the Class Visualizer tool was able to detect

It can be noticed that the amount of nested classes is the same in both applications' results. Class Visualizer does not differentiate between aggregation and composition. Both of these types of associations are labeled as "Associations (Uses)". The remaining associations detected by the proposed application are also found in the "Associations Usages (Used By)" branch of the "Browser" tab. All of these classes are detected in the proposed application's outputs too, however, the Class Visualizer tool has missed out on 5 class associations. The missed associated classes are Account, Transaction, BalanceInquiry, Withdrawal, and Deposit.

When it comes to the encapsulation outputs, it can be seen that the "username" attribute's encapsulation is done with an almost textbook-like implementation. The code listing 2 shows the relevant part of code that encapsulates the said attribute.

Listing 2: Encapsulation of an attribute

```
public class Account
{
    private String username;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
    //...
}
```

After a short inspection of the "ATM" class' body, it can be noticed that the "Users" attribute of "Iterator" type has no access modifier, and thus the class breaks the encapsulation principles.

Lastly, "polymorphism" part of the output will be discussed. Since none of the tools mentioned in the second chapter detect polymorphism, it has to be evaluated by hand. After searching for an instance of the "currentAccount" object in the "ATM" class' body, the following lines that implement polymorphism can be found 3.

Listing 3: Polymorphic object behaviour

```
public class Account
{
   public int getAccountNumber()
   {
      return accountNumber;
   }
   // ...
}

public class AccountFactory extends Account {
   // ...
}

public class BankDatabase
{
   public Account getAccount(int accountnumber)
   {
      for (Account currentAccount : accounts)
      {
         if (currentAccount.getAccountNumber() == accountnumber)
            return currentAccount;
      }
      return null;
   }

   // ...
}
```

It can also be mentioned that the application has detected the implementation of the Singleton design pattern. This design pattern could supposedly be detected by the tool proposed in [11], however inputting the same ".class" files from the same folder that was inputted to the Class Visualizer tool led to no output. After a quick search for the mentioned method, the following code can be found 4. It shows a fairly standard way of the pattern's implementation. Furthermore, the method's name greatly hints that such a pattern is put to use.

Listing 4: An ATM class method implementing the Singleton design pattern

```
public class ATM
{
   private static ATM uniqueinstance;

   public static ATM getinstance() {
        if (uniqueinstance == null) {
                uniqueinstance = new ATM();
        }
        return uniqueinstance;
   }

   // ...
}
```

All the outputs have now been discussed and thus the results have been evaluated. The next section wraps up what all these efforts led to.

5. Conclusions and further work

Getting a grasp on the utilization of object-oriented mechanisms in code is hard when a programmer has no experience writing code in object-oriented languages. General definitions with real-life examples are a good way to get an idea of what these mechanisms represent, but assigning a strict, code-based definition to them can be very helpful. The proposed solution attempts to aid programmers further by providing them feedback on their usage of said mechanisms in their code instantly. Some of these mechanisms have special keywords assigned to them and thus are easy to spot in code, but the ones that do not require larger context. This often means that a number of conditions have to be met for the mechanism to be implemented correctly. Or that the user has to know about all the classes present in the source code for them to spot these mechanisms. Even though this paper named all the conditions, spotting them in code can still be challenging. The application aims to help with this, which should help its users be more confident in their utilization of these mechanisms that lead to better-structured code.

The tool was evaluated with the use of code examples found online, open source-code projects and projects made by the students of the Faculty of Informatics and Information Technologies at Slovak Technical University. The students reported a number of issues which were all taken care of in order to improve the tool.

The application has room to grow - detection of other, yet untouched design patterns could be added, and already implemented features could be improved. Continuing in the set approach to mechanism detection from different coding languages' source codes would make the application accessible to more programmers. Adding an option to read source code from an online repository (for example GitHub) for analysis could also improve the user's experience and convenience. Rewriting the application in Java in order to have it distributed as a JAR platform-independent tool is yet another way of making the application accessible to more users. Adding hints on how to improve the read code based on information gathered by the application would also be very helpful for the application's users.

Recently the field of artificial intelligence has expanded greatly, and thus finding more AI-driven tools and comparing their features to ours could prove to be beneficial. Comparing this application's accuracy with more tools and testing them on larger, more complex open-source projects could lead further improvements of the application. Since the tool was tested on smaller projects, mostly written by students, it is currently unclear how much time the application takes to analyze much larger projects. Students' projects were around 1 500 lines of code long on average and could be analyzed in a couple of seconds at worst. Testing the application on such projects could also lead to valuable information and metrics about the tool.

Acknowledgment

This article was written thanks to the generous support under the Operational Program Integrated Infrastructure for the project: "Support of research activities of Excellence laboratories

STU in Bratislava", Project no. 313021BXZ1, co-financed by the European Regional Development Fund and as part of the national project "Increasing Slovakia's resistance to hybrid threats by strengthening public administration capacities", project code ITMS2014+: 314011CDW7 - this project is supported by the European Social Fund and the Operational Program Integrated Infrastructure for the project: National infrastructure for supporting technology transfer in Slovakia II – NITT SK II, co-financed by the European Regional Development Fund.

References

[1] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Connallen, K. A. Houston, Object-oriented analysis and design with applications, ACM SIGSOFT software engineering notes 33 (2008) 29–29.

[2] V. Vranić, Objektovo-orientované programovanie: Objekty, java a aspekty, Vydavatel'stvo STU (2008).

[3] B. Dathan, S. Ramnath, A. Approach, S. Edition, Object-Oriented Analysis, Design and Implementation, Springer, 2015.

[4] B. P. Douglass, Real-time design patterns: robust scalable architecture for real-time systems, Addison-Wesley Professional, 2003.

[5] N. Shi, R. A. Olsson, Reverse engineering of design patterns from java source code, in: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), IEEE, 2006, pp. 123–134.

[6] M. Wyrich, J. Bogner, Towards an autonomous bot for automatic source code refactoring, in: 2019 IEEE/ACM 1st international workshop on bots in software engineering (BotSE), IEEE, 2019, pp. 24–28.

[7] Y. Gil, M. Orrù, The spartanizer: Massive automatic refactoring, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017, pp. 477–481. doi:10.1109/SANER.2017.7884657.

[8] M. G. Al-Obeidallah, M. Petridis, S. Kapetanakis, A survey on design pattern detection approaches, International Journal of Software Engineering (IJSE) 7 (2016) 41–59.

[9] R. MOREIRA, E. FERNANDES, E. FIGUEIREDO, based comparison of design pattern detection tools (2022).

[10] R. S. Rao, A review on design pattern detection, Int J Eng Res Technol 8 (2019) 756–762.

[11] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. T. Halkidis, Design pattern detection using similarity scoring, IEEE transactions on software engineering 32 (2006) 896–909.

[12] M. Fowler, Refactoring: Improving the design of existing code, in: 11th European Conference. Jyväskylä, Finland, 1997.

[13] E. Murphy-Hill, C. Parnin, A. P. Black, How we refactor, and how we know it, IEEE Transactions on Software Engineering 38 (2011) 5–18.

[14] Z. Troníček, Refactoringng: A flexible java refactoring tool, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing, 2012, pp. 1165–1170.