Class functionality and its related concepts: research and practice

Alexander Prutzkow

***Abstract*—A class functionality in object-oriented programming is a part of the application programming interface (API). We reveal concepts related to the class functionality. The functionality includes an abstract functionality – a set of abstract methods. The main characteristic of the functionality is its stability. The functionality of a new version must ensure backward compatibility. Version numbers are used to indicate backward compatibility or incompatibility. Backward compatibility can be source, binary, or functional. The evolution of the functionality consists of its modification or extension. In case of breaking changes, a good practice is to mark old methods as deprecated without removing. The evolution of the abstract functionality differs from the evolution of the class functionality. Adding or removing a method in the abstract functionality is backward-incompatible. Depending on the change of the functionality in subclasses, inheritance can be functionally extendible or functionally overridable. We introduce a pattern style "Abstraction Raising" to improve stability. The style consists of using more abstract items. We identified such items for a type, variable, field, class, and constructor. We demonstrate an example of using the pattern style when changing the return type. We investigate relationships between the class functionality and the considered concepts in practice. We survey scientific articles with statistical data characterizing the relationships.**

***Keywords*—API, functionality, evolution, compatibility.**

I. Introduction

The concept of reuse involves developing a program based on existing software modules or systems [1]. Software modules or systems can have different representations, including libraries (see examples of libraries in [2–3]). The removal of program fragments into libraries eliminates code duplication [4]. Classes in object-oriented programming using libraries we call client classes. Libraries provide access to methods for solving problems through an application programming interface (API).

II. Motivation, the Purpose of the Study, and the Article Organization

The core of the API is a class functionality. The functionality is related to various concepts. These relationships are not clearly described, which makes it difficult to understand the functionality itself and its evolution.

The purpose of the study is to identify concepts related to the functionality and, based on this, to improve the stability of the functionality by introducing a pattern style.

In Part 1 of this article, we explore the concepts related to the class functionality. In Part 2, we introduce the "Abstraction Raising" pattern style that allows making breaking changes non-breaking. In Part 3, we survey statistical data that characterize the relationships between the class functionality and the discussed concepts.

III. Concept Map

We explore the concepts related to the functionality and depict them as a concept map (fig. 1).

Let's discuss these concepts in details.

IV. What is a class functionality

A class in object-oriented programming includes fields, constructors, and methods. The values of the fields of a class object determine the state of the object. Class methods inherited by subclasses or intended for calling by objects of other classes constitute the functionality of the class. We name the set of class methods with the public modifier as a public functionality.

The functionality is used by client classes to solve their problems.

The main characteristic of the functionality is stability. The stability of the functionality is its ability to be unchanged when the class changes (based on [5]).

V. Abstract functionality

There are some cases when different classes must have the same functionality:

- performing the same actions on objects of different classes; for example, comparing, reading, and writing objects;
- raising the level of abstraction of created objects and hiding their classes from a client class.

In these cases, an abstract functionality is used. The abstract functionality is a functionality that defines abstract methods implemented by other classes. An abstract method is a method without a body.

Interfaces and abstract classes define the abstract functionality in the Java programming language.

Manuscript received September 6, 2024.

A. Prutzkow is with the Ryazan State Radio Engineering University, 390005, Gagarin str., 59/1, Ryazan, Russia, and with Lipetsk State Pedagogical University, 398020, Lenin str., 42, Lipetsk, Russia (e-mail: mail@prutzkow.com).

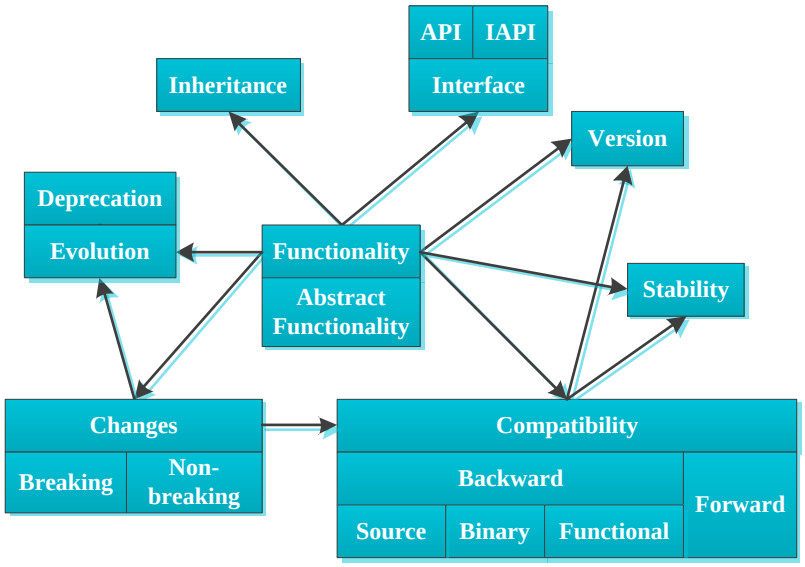

Fig. 1. Class functionality and related concepts

The abstract functionality can be divided into the following types depending on how it is used by client classes:

- external: client classes implement an abstract functionality; for example, the Comparator interface;
- internal-external: client classes use objects of classes that implement an interface, but don't create such objects; for example, the *java.sql.Connection* interface;
- internal: not accessible for client classes.

VI. Functionality is not a functional

The class functionality should be distinguished from a functional. The functional is a mapping of a number to a function [6].

VII. Interface, API, and IAPI

Interface is an access point to a component that client systems can reference to reuse functionalities [7].

We've explored other meanings of the “interface” term in [8].

API include the collective public functionality of the classes that constitute the library.

The characteristics of an API are stability, maintainability, and documentation [7].

Internal API (IAPI) of a library is the collective functionality of the classes that constitute the library, used by objects of the classes of the library, but not by client classes.

Different access modifiers are used for these interfaces:

- API – public;
- IAPI – protected, by default.

Modifiers in the Java language allow you to restrict access to a class, subclass, or package. However, there is no easy way to restrict access to only a specific part of a package [9]. A common practice is to make package classes public and use a naming convention that appends the word “internal” [4, 7] to the name of the internal package (e.g., the *org.elasticsearch.client.internal* package) or the @Internal annotation [9].

VIII. Another definition of the concept of API

API also refers to a set of commands and their parameters for working with web services. We call this API as web API, but the details are beyond the scope of this article. Some aspects discussed in this article are also relevant for the web API. More about web API, its life cycle, REST and GraphQL architectural styles can be read in [10–14].

IX. Software changes

The extendible or incorrectly designed functionality of a class changes. According to ISO/IEC 14764, software changes are divided into four types (see also [15]) with examples:

- corrective – when errors are detected in the software;
- adaptive – when migrating to a new operating system;
- perfective – improvement of software characteristics;
- preventive – refactoring of a program or part of it to improve its maintenance.

These changes are related as follows (fig. 2) [16]. The arrows in the figure mean “lead to”.

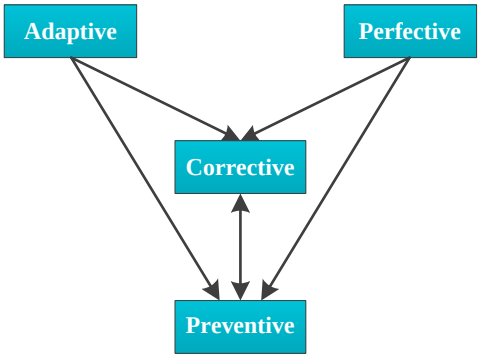

Fig. 2. Relationships between types of changes

There are other classifications of software changes [17].

X. BACKWARD AND FORWARD COMPATIBILITY

When changing the functionality, it is necessary to ensure backward compatibility so that client classes don't have to be changed.

Backward compatibility is the property of a class (library) to maintain the functionality of the program when the old version of the class (library) is replaced by a new version.

Forward compatibility is the property of a class (library) to maintain the functionality of the program when a new version of the class (library) is replaced by an old version.

Ensuring backward compatibility is one of the challenges of designing the functionality and APIs.

There are the following types of backward compatibility of a class (library) (based on [18]):

- source compatibility – a program compiled with an old version of a class (library) is compiled with a new version of the class (library);
- binary compatibility – a program compiled with an old version of a class (library) works with a new version of the class (library);
- functional compatibility – a program compiled with a new version of a class (library) produces the same result as with the old version of the class (library).

Binary compatibility doesn't imply source compatibility. Consider the example from [19] (listing 1). When replacing the compiled version of a class (lines 1-6) to a new one (lines 8-13) and running the already compiled ClientRunner class, no error will occur. When recompiling the ClientRunner class, a compilation error will occur.

Listing 1. An example of binary compatibility and source incompatibility

```
1  // class version 1
2  public class Foo {
3      public static java.util.List<String> foo() {
4          return new java.util.ArrayList<String>();
5      }
6  }
7
8  // class version 2
9  public class Foo {
10     public static java.util.List<Integer> foo() {
11         return new java.util.ArrayList<Integer>();
12     }
13 }
14
15 // client program using the class
16 public class ClientRunner {
17     public static void main(String[] args) {
18         java.util.List<String> list = Foo.foo();
19         System.out.println(list.size());
20     }
21 }
↳
```

XI. VERSION, BACKWARD AND FORWARD COMPATIBILITY

Semantic versioning (*semver*) of classes is used to demonstrate backward compatibility.

The designation of a stable version is as follows:

X.Y.Z,

where X is the major version number; Y is the minor version number; Z is the patch number.

The rules for assigning version designations are as follows [4]:

- if classes (libraries) have different major version numbers, then no compatibility is guaranteed; for example, versions 2.13.4 and 3.0.2 may be completely incompatible;
- if classes have the same major version number, but different minor version numbers, then the class with the higher minor version number must be backward-compatible with the class with the lower minor version number; for example, a class with version 2.13.4 must be backward-compatible with a class of version 2.5.3;
- if a class has the same major and minor version numbers, then they must be forward- and backward-compatible.

XII. EVOLUTION OF FUNCTIONALITY

A. Definition, reasons and basic approach

Evolution is a process of giving a class (library) new or improving existing characteristics by changing this class (library).

The reasons for the API evolution, and with it the functionality, are introducing of new functions and the need to improve quality (usability and maintainability) [20, 21].

Some changes in a class are breaking. A way to smooth out such changes for client classes is to mark the methods that will be removed as deprecated. Typically, methods are removed in the class with the new major version number.

B. Breaking and non-breaking changes

Changes may be breaking or non-breaking.

Non-breaking changes include [22]:

- add method;
- pull up method;
- gain visibility;
- remove final modifier;
- add static modifier;
- depreciate method;
- extract method.

Breaking changes include [22]:

- remove method;
- lost visibility;
- change in return type;
- change in parameter list;
- change in exception list;
- add final modifier;
- remove static modifier;
- move method;
- rename method;
- push down method;
- inline method.

A detailed description of some of the listed non-breaking changes with examples can be found in [23]. These changes are called structural transformations. In [23] also breaking behavioral modifications are listed:

- new method contract – change of preconditions or postconditions of method execution, for example, for an input parameters a requirement of null inequality appears;
- implement new interface – an implementation of a new interface is added to the class;
- changed events order – here the event is a situation, but which the client object can react to;
- new enumeration constant – adding a new constant to the enumeration.

Breaking changes are classified in [19].

C. Breaking changes and deprecated methods

The general approach to introducing breaking changes involves three steps (introduce – deprecate – remove):

- introducing a new version of a method without deleting its old version;
- marking the old version of the method as deprecated;
- removing the old version of a method when moving to a new version of a class.

There are the following reasons to make a method deprecated [24] with examples:

- avoid bad coding practices – use a constructor instead of a method for setting a field value;
- design pattern – create an object using the “Builder” pattern [25];
- dissent usage – an interface method that performs default actions;
- functional defects – a method that doesn't perform actions correctly;
- merged to existing method – adding new actions (such as checking values) to a method makes another method unnecessary;
- new feature introduced – a method replaced by a new method;
- no dependency support – a method that depends on a method that no longer exists;
- redundant method – the method is not called;
- renaming of feature – the method name is inaccurate or doesn't comply with the new naming convention;
- security flaws – the method causes a vulnerability in the program;
- separation of concerns – instead of a method, several new methods are used;
- temporary feature – the method is introduced to solve temporary problems.

TABLE 1. AWT AND SWING LIBRARY DEPRECATED METHODS AND THEIR SHARES

Reason	AWT, %	Swing, %
Conformity to naming conventions	50.0	24.4
Simplification	16.7	4.9
Introduction of new concepts and classes	11.4	4.9
Reducing coupling	1.8	7.3
Encapsulation	2.6	0
Conformity to supertype contracts	1.8	7.3
Deprecation without replacements	5.3	34.1
Redesign of existing features	10.5	17.1

The most frequent reasons to make a method deprecated are new feature introduced, functional defects, design pattern [24].

In [26] the AWT and Swing libraries are examined. The following reasons for making a method deprecated and their share are classified (table 1).

D. When are breaking changes good?

Let's say a class provides a connection between two network nodes via a secure protocol. After some time, vulnerabilities of this protocol were revealed and a decision was made to switch to a new protocol. The new class doesn't provide backward compatibility, forcing developers of client classes to modify them and switch to a new protocol, eliminating vulnerabilities. In this case, breaking changes improve the security of the program.

XIII. EVOLUTION OF ABSTRACT FUNCTIONALITY

A. Differences from the functionality evolution, techniques for non-breaking changes

The evolution of the external abstract functionality differs from the evolution of the class functionality: changes in the abstract functionality are backward-incompatible. The reason is that the abstract functionality strictly defines the functionality of classes, and changing the abstract functionality involves changing the functionality of the classes. The evolution of the internal-external abstract functionality is the same to the class functionality.

There are techniques to make changes to the abstract functionality non-breaking. We demonstrate the techniques using two changes as an example: adding and removing a method. These operations can be considered more broadly. Changing a method can be considered as removing an old method and adding a new method.

B. Adding a method

Adding a method with an implementation is non-breaking. Adding an abstract method is breaking and requires a default implementation for the abstract class (listing 2) or interface (listing 3) [27].

Listing 2. Abstract class with added method

```
1  public abstract class AbstractClass {
2      public abstract void method();
3
4      public void newMethod() {
5          System.out.println("AbstractClass is working v2");
6      }
7  }
↳
```

Listing 3. Interface with added method

```
1  public interface Interface {
2      void method();
3
4      default void newMethod() {
5          System.out.println("Interface is working v2");
6      }
7  }
↳
```

The default implementation may be to throw an UnsupportedOperationException [25, 27, 28, 29] (listing 4 [25]).

Listing 4. UnsupportedOperationException thrown as default implementation

```
1  @Override
2  public V setValue(V value) {
3      throw new UnsupportedOperationException();
4  }
↳
```

This exception alerts the programmer that a method is implemented by default in an abstract class or interface, but not in the client class.

The default implementation of throwing UnsupportedOperationException is criticized in [18] for requiring the addition of a try – catch statement to catch the exception, which increases the program length. The default implementation of doing nothing creates backward incompatibility when the implementation is called [4].

Another way to add a method that doesn't require a default method implementation is to create a subinterface with a new method [30] (listing 5).

Listing 5. An interface inheriting superinterface methods and adding a new method

```
1  public interface InterfaceWithNewMethod extends Interface {
2      void newMethod2();
3  }
↳
```

C. *Removing a method*

Let's say there is Interface with removingMethod that needs to be removed (listing 6). Interface is not a subinterface.

Listing 6. Original interface with a deleted method

```
1  public interface Interface {
2      void method();
3      void removingMethod();
4  }
↳
```

Let's introduce SuperInterface and move all the remaining methods to it (listing 7, lines 1-3). Let's make Interface as a subinterface of SuperInterface (lines 5-7).

Listing 7. Interfaces after method removal

```
1  public interface SuperInterface {
2      void method();
3  }
4
5  public interface Interface extends SuperInterface {
6      void removingMethod();
7  }
↳
```

New versions of classes must implement SuperInterface, not Interface.

D. *Interface-segregation principle*

When changing interfaces, it is worth remembering the interface-segregation principle [31]:

Clients should not be forced to depend on methods that they don't use.

XIV. Functionality and inheritance

Inheritance can be divided into two types depending on the impact on the functionality of the subclass [5]:

(1) Functionally extendible (“inheritance for the sake of functionality”), when a subclass gets the functionality of a superclass and extends it with its own. Examples of this type of inheritance are custom exception subclasses of the Exception class or subclasses of the JFrame class of the Swing graphics library.

(2) Functionally overridable (“inheritance for the sake of polymorphism”), when the methods of the superclass are overridable in the subclass, and the functionality in the subclass doesn't change. Examples of this type of inheritance are input-output stream classes. These classes have the same functionality, but allow reading and writing data from different sources: files, strings, arrays. The same functionality allows you to replace reading from a file with reading from a string. The overridable functionality is defined by an abstract functionality.

Subclasses can be created for different purposes. In the case of functionally extendible inheritance, subclasses are created to access the functionality of the class hierarchy and add new functions to it, and in the case of functionally overridable inheritance, they are created to override the functionality of the superclass to solve problems with new specifics.

In both types of inheritance, subclasses must be more specialized (less general) than the superclass.

Design and document for inheritance or else prohibit it [25].

XV. Pattern style “Abstraction raising”

A. *How to improve the stability of the class functionality?*

Developers must make headers of the methods stable, since changing them almost always involves binary compatibility [19]. As a result of studying the class functionality and the related concepts, it becomes necessary to find an approach to improve the stability of the functionality.

B. *Concept*

A pattern style is a concept intended to be used further in design patterns.

We introduce the “Abstraction Raising” pattern style. Only what is hidden can be changed without risk [32]. The pattern style consists in using more abstract items (table 2; based on [18, 25, 33, 34]) to improve the stability of the class functionality. We define a factory as a class with a method that creates an object of another class. We define a factory method as a static method of a class that creates an object of the same class (static factory method in [25]).

Table 2. Items and their more abstract ones

Item	More Abstract Item
byte, short, int, long, float, double, char	Class (not a wrapper class: Byte, Short, etc)
boolean	Enum
Field	Method
Class	Interface or abstract class
Constructor	Factory or factory method

Using more abstract items allows the following:

- hide specific elements from client classes behind their abstractions for different purposes (prohibition of creation of class objects, substitution of objects of some classes with objects of other classes, etc.);
- make breaking changes non-breaking.

Whether or not to raise abstraction depends on the

situation in which it is used.

Improving stability in the pattern style is achieved by complicating of the program.

We demonstrate how to improve the stability of the functionality via the example of a breaking change converted into non-breaking one.

C. Non-breaking change of return type

Let the method return a value of type int. When designing the method of OnlyClass (listing 8), a more abstract return type was used – the MethodResult class (listing 9).

Listing 8. OnlyClass class with method version 1

```
1  public final class OnlyClass {
2      public MethodResult method() {
3          System.out.println("OnlyClass is working v1");
4          int originalResult = 1;
5          MethodResult result = new MethodResult(originalResult);
6          return result;
7      }
8  }
↳
```

Listing 9. MethodResult class version 1

```
1  public final class MethodResult {
2      private int integer;
3
4      MethodResult(int integer) {
5          this.integer = integer;
6      }
7
8      public int getInt() {
9          return this.integer;
10     }
11 }
↳
```

The client class has the form (listing 10).

Listing 10. Client class

```
1  public class ClientRunner {
2      public static void main(String[] args) {
3          OnlyClass onlyClass = new OnlyClass();
4          MethodResult result = onlyClass.method();
5          System.out.printf("Integer result: %d\n", result.getInt());
6      }
7  }
↳
```

In version 2 of method, it turned out that the return type must be double. In MethodResult, the type and field name were changed (listing 11, line 2), getDouble has been added (lines 8-10), in getInt the return of the integer type field value has been replaced by rounding the value of a field with the double type (line 13).

Listing 11. MethodResult class version 2

```
1  public final class MethodResult {
2      private double doubleValue;
3
4      MethodResult(double doubleValue) {
5          this.doubleValue = doubleValue;
6      }
7
8      public int getInt() {
9          return (int) Math.round(this.doubleValue);
10     }
11
12     public double getDouble() {
13         return this.doubleValue;
14     }
15 }
↳
```

OnlyClass was changed as well (listing 12).

Listing 12. OnlyClass class version 2

```
1  public final class OnlyClass {
2      public MethodResult method() {
3          System.out.println("OnlyClass is working v2");
4          double originalResult = 1.2;
5          MethodResult result = new MethodResult(originalResult);
6          return result;
7      }
8  }
↳
```

However, the client class (listing 10) remained unchanged, since method returned an object of the modified MethodResult class, not the value of the primitive type.

This change will be source-compatible. The change will be functional-compatible if the rounding in method of the version 2 is equivalent to the integer value produced by method of the version 1.

D. Non-breaking change to formal parameter list

A non-breaking change to the formal parameter list is to use the MethodParameters data class with the method parameters. Instead of parameters, an object of MethodParameters is passed to the method. This allows changing the formal parameter list without modifying the method header. The example classes are bulky. You can see the example classes and run the program, as well as the classes of other examples, by downloading the program project from the website http://prutzkow.com [35].

XVI. Class functionality and related concepts in practice

A. Preliminary note

We survey statistic studies characterizing the relationships of the API, and therefore the functionality, with the related concepts.

B. API, changes, deprecated API

McDonnell T. et al. [36] explored the API versions 3–15 (2009–2011) of the Android operating system:

- in every API version, on average 149 classes and 158 methods were changed, 37 methods were added, 2 methods were removed, 179 fields were changed, 32 fields were added, and fields were not removed;
- on average, 28% of client methods call deprecated API methods; 50% of API method calls remain unchanged

for 16 or more months after these methods were declared deprecated.

C. Deprecated API

Zhou J. and Walker R.J. [37] examined comments of deprecated APIs in 26 Java projects:

- only 55% of comments had an indication of the API, which is a replacement for the deprecated API;
- on average 9.1% of comments contained reasons to make the API deprecated;
- in 12 projects, the API was initially declared as deprecated, and in the next versions this declaration was removed;
- in 3 projects the API was removed and then restored as deprecated (remove – resurrect – deprecate);
- in 15.4% of projects, the deprecated API was completely removed in the following versions, in 30.8% it was partially removed, in 53.8% it was not removed;
- 74.4% of removed deprecated APIs were removed during a major version release.

D. API, breaking and non-breaking changes

Xavier L. et al. [38] studied changes in 317 libraries. Changes were classified into breaking and non-breaking, as well as types, fields, and methods (table 3). The most frequently used classes and interfaces were identified (table 4).

TABLE 3. SHARE OF TOTAL CHANGES AND BREAKING CHANGES BY CLASS ELEMENT

Element	Total, %	Breaking changes, %
Types	12.3	18.9
Fields	13.4	37.4
Methods	74.3	27.8
All	100	28.0

TABLE 4. THE MOST FREQUENTLY USED CLASSES AND INTERFACES OF THE JAVA PROGRAMMING LANGUAGE

Class	Number of client program classes
java.util.ArrayList	143 454
java.io.IOException	136 058
java.util.List	134 053
java.util.HashMap	94 220
java.io.File	88 703

Brito A. et al. [22] analyzed changes in two graph display and image loading libraries and presented the following statistics:

- methods are changed more often than fields and types during breaking and non-breaking changes;
- the most common breaking changes are method deletion (44%) and non-breaking changes are method addition (67%).

E. API, breaking changes, version

Ochoa L. et al. [9] analyzed pairs of library versions with breaking changes in two sets (tables 5–6) and compared the obtained results with the results from [39] (table 7).

TABLE 5. SHARES OF TOTAL AND BREAKING CHANGES IN LIBRARIES BY VERSION TYPE (SET 1)

Version	Share of changes, %	Share of breaking changes, %
Major	2.4	72.7
Minor	23.2	50.1
Patch	74.4	24.2
	100.0	

TABLE 6. SHARES OF TOTAL AND BREAKING CHANGES IN LIBRARIES BY VERSION TYPE (SET 2)

Version	Share of changes, %	Share of breaking changes, %
Major	2.4	61.8
Minor	23.1	37.9
Patch	74.5	14.7
	100.0	

TABLE 7. SHARES OF TOTAL AND BREAKING CHANGES BY VERSION TYPE [39]

Version	Share of changes, %	Share of breaking changes, %
Major	14.8	35.9
Minor	37.2	35.7
Patch	48.1	23.8
	100.0	

Mostafa S. et al. [2] tested 68 pairs of versions of 15 libraries. They found 76.5% of pairs contained behavioral backward incompatibilities.

Dietrich J. et al. [19] found that 75% of version pairs of Java programs had breaking changes. Only 2 out of 109 programs had a stable API.

F. API, changes, IAPI

Hora A., Robbes R. et al. [40] examined the environment of a programming language with more than 3 500 client programs and identified 118 API changes:

- 50% of methods added with keeping of the old method; 50% of methods added with removal of the old method;
- 8% of changes are related to the IAPI;
- 53% of changes caused changes in client programs;
- propagation time is shorter from the 2nd quartile for adding a method while removing an old method (a breaking change) than for adding a method while keeping an old method (a non-breaking change) (table 8).

TABLE 8. DISTRIBUTION OF TIME FOR MAKING CHANGES TO CLIENT PROGRAMS, DAYS

	1st quartile	2nd quartile	3rd quartile	4th quartile
Adding a method with deleting the old method	16	18	201	211
Adding a method while keeping the old method	7.5	121	334.5	662

Hora A., Valente M.T. et al. [7] studied IAPI in 5 Java programs and concluded that:

- 23.5% of client programs depend on the IAPI of Eclipse;
- 7% of IAPI in new versions became API.

Mastrangelo L. et al. [41] found that 25% of artifacts used an undocumented Java IAPI class, which can violate the

security of program execution.

Dietrich J. et al. [19] as a result of their study recommend the following:

- developers should use a naming convention to distinguish between public and private packages; this will allow to identify calls to private packages automatically;
- you should not rely on non-public JRE packages (sun.* etc.), as these packages may not be available on some platforms, such as Android.

G. Automated API creation

Reimann L. and Kniesel-Wünsche G. [42] introduced adaptering. Adaptering consists of using the “Adapter” pattern [34] to create an API of a new library. The new library calls functions of the old library. The authors identified API changes that can be made automatically and developed a program for manual changes.

XVII. CONCLUSION

In our study, we investigated the relationships between the class functionality and its evolution, changes, compatibility, API, inheritance, version, and stability. We verified the existence of these relationships in practice with statistical data. Our research will be useful:

- scientists when planning their experiments in the field of class and API design;
- developers when designing the functionality of a class, taking into account its relationships with other concepts.

The introduced pattern style makes the functionality of the class more stable, and therefore the API in which this class is used. However, using the style complicates the program.

Scientific articles and papers on the API evolution from 1994 to 2018 were surveyed in [43].

REFERENCES

[1] Santana de Almeida E. Software Reuse and Product Line Engineering. In Handbook of Software Engineering, 2019:321–348. DOI: 10.1007/978-3-030-00262-6_8.

[2] Mostafa S. et al. Experience Paper: A Study on Behavioral Backward Incompatibilities of Java Software Libraries. In 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2017:215–225. DOI: 10.1145/3092703.3092721.

[3] Finlayson M.A. Java Libraries for Accessing the Princeton Wordnet: Comparison and Evaluation. In the 7th International Global WordNet Conference (GWC), 2014:78–85.

[4] Lelek T., Skeet J. Software Mistakes and Tradeoffs. How to Make Good Programming Decisions. Manning, 2022.

[5] Prutzkow A.V. Tonkosti Programmirovanija v Primerakh [Programming Subtleties in Examples]. Kurs, 2022. [in Rus].

[6] Hersh R. What is Mathematics, Really? Oxford University Press, 1997.

[7] Hora A., Valente M.T. et al. When Should Internal Interfaces Be Promoted to Public? In 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), 2016:280–291.

[8] Prutzkow A.V. Printsipy Razrabotki Programmnyh Interfejsov v Industrialnyh Informatsionno-Izmeritelnyh i Upravljajuschih Sistemah [Principles for Development of Program Interfaces in Industrial Information, Measuring, and Controlling Systems]. In Kontrol'. Diagnostika, 2021, 24(10):44–47. [in Rus]. DOI: 10.14489/td.2021.10.pp.044-047.

[9] Ochoa L. et al. Breaking Bad? Semantic Versioning and Impact of Breaking Changes in Maven Central: An External and Differentiated Replication Study. In Empirical Software Engineering, 2022, 27(3):61.

[10] De B. API Management: An Architect’s Guide to Developing and Managing APIs for Your Organization, 2nd ed. Apress, 2023. DOI: 10.1007/979-8-8688-0054-2.

[11] Geewax J.J. API Design Patterns. Manning, 2021.

[12] Medjaoui M. et al. Continuous API Management, 2nd ed. O’Reilly, 2021.

[13] Weir L. et al. Enterprise API Management. Packt, 2019.

[14] Zimmermann O. et al. Patterns for API Design. Addison-Wesley, 2023.

[15] Kim M. et al. Software Evolution. In Handbook of Software Engineering, 2019:223–284. DOI: 10.1007/978-3-030-00262-6_6.

[16] Grubb P., Takang A.A. Software Maintenance: Concepts and Practice. World Scientific, 2003.

[17] Chapin N. et al. Types of Software Evolution and Software Maintenance. In Journal of Software Maintenance and Evolution: Research and Practice, 2001, 13(1):3–30.

[18] Tulach J. Practical API Design: Confessions of a Java Framework Architect. Apress, 2008.

[19] Dietrich J. et al. Broken Promises: An Empirical Study into Evolution Problems in Java Programs Caused by Library Upgrades. In Software Evolution Week – IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR–WCRE), 2014, 1:64–73.

[20] Stocker M., Zimmermann O. From Code Refactoring to API Refactoring: Agile Service Design and Evolution. In SummerSOC 2021, CCIS 1429, 2021:174–193. DOI: 10.1007/978-3-030-87568-8_11.

[21] Granli W. et al. The Driving Forces of API Evolution. In IWPSE, 2015. DOI: 10.1145/2804360.2804364.

[22] Brito A. et al. APIDiff: Detecting API Breaking Changes. In IEEE 25th International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2018:507–511.

[23] Dig D., Johnson R.E. How Do APIs Evolve? A Story of Refactoring. In Journal of Software Maintenance and Evolution, 2006, 18(2):83–107.

[24] Sawant A.A. et al. Why are Features Deprecated? An Investigation into the Motivation behind Deprecation. In IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018:13–24. DOI: 10.1109/ICSME.2018.00011.

[25] Bloch J. Effective Java, 3rd ed. Addison-Wesley, 2018.

[26] Hou D., Yao X. Exploring the Intent behind API Evolution: A Case Study. In 18th Working Conference on Reverse Engineering, 2011:131–140.

[27] Schildt H. Java. The Complete Reference, 9th ed. McGraw-Hill, 2014.

[28] Blinov I., Romanchik V.S. Java from. Chetyre Chetverti, 2020. [in Rus].

[29] Sharan K. Beginning Java 9 Fundamentals: Arrays, Objects, Modules, JShell, and Regular Expressions. Apress, 2017. DOI: 10.1007/978-1-4842-2902-6.

[30] Spoon A. Anti-Deprecation: Towards Complete Static Checking for API Evolution. In 2nd International Workshop on Library-Centric Software Design (LCSD), 2006:65–74.

[31] Martin R. Agile Software Development. Principles, Patterns, and Practices. Prentice Hall, 2003.

[32] Endres A., Rombach D. A Handbook of Software and Systems Engineering. Empirical Observations, Laws, and Theories. Pearson, 2003.

[33] Parnas D.L. On Criteria to Be Used in Decomposing Systems into Modules. In Communications of the ACM, 1972, 15(12):1053–1058.

[34] Gamma E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[35] Prutzkow A.V. Internet-Resurs dlja Razmeschenija Rezultatov Nauchnoj i Obrazovatelnoj Dejatelnosti [Internet-Resource for Scientific and Educational Work Result Publishing]. In Vestnik of the RSREU, 2018, 63:84–89. [in Rus]. DOI: 10.21667/1995-4565-2018-63-1-84-89.

[36] McDonnell T. et al. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In IEEE International Conference on Software Maintenance, 2013:70–79.

[37] Zhou J., Walker R.J. API Deprecation: A Retrospective Analysis and Detection Method for Code Examples on the Web. In 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2016:266–277.

[38] Xavier L. et al. Historical and Impact Analysis of API Breaking Changes: A Large-Scale Study. In IEEE 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2017:138–147.

[39] Raemaekers S. et al. Semantic Versioning and Impact of Breaking Changes in the Maven Repository. In Journal of Systems and Software, 2017, 129:140–158. DOI: 10.1016/j.jss.2016.04.008.
[40] Hora A., Robbes R. et al. How do Developers React to API Evolution? A Large-Scale Empirical Study. In Software Quality Journal, 2016. DOI: 10.1007/s11219-016-9344-4.
[41] Mastrangelo L. et al. Use at Your Own Risk: the Java Unsafe API in the Wild. In International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2015.
[42] Reimann L., Kniesel-Wünsche G. Adaptoring: Adapter Generation to Provide an Alternative API for a Library. In International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2024.
[43] Lamothe M. et al. A Systematic Review of API Evolution Literature. 2020. DOI: 10.1145/1122445.1122456.