UML Representation of Object-Oriented Design Antipatterns

Lidiya S. Ivanova
Higher IT School
National Research Tomsk State University
Tomsk, Russia
lidiya.ivanova@persona.tsu.ru

Danila A. Sokolov
Higher IT School
National Research Tomsk State University
Tomsk, Russia
danila.a.sokolov@gmail.com

Oleg A. Zmeev
Higher IT School
National Research Tomsk State University
Tomsk, Russia
ozmeyev@gmail.com

***Abstract*—Nowadays the ability to apply, implement and modify patterns of design and architecture has become a one of primary skills for software engineers. Competence of pattern design and implementation involves detecting and correcting inefficient solutions known as antipatterns. However, unlike patterns, very few antipatterns have a graphical representation so that an inefficient solution to a specific problem can be detected visually and refactored. Detecting antipatterns is not simple even with full set of technical documentation. This paper proposes a graphical UML representation of antipatterns to detect them at various stages of the software lifecycle. It proposes a method to refactor described antipatterns to improve software design quality and avoid software development process risks. UML diagrams modeling of 18 antipatterns is presented and refactoring method for all of them was described. Most of antipatterns were diagrammed using information from text descriptions and additional notes about arguable properties of antipatterns were included.**

Keywords— antipattern, class diagram, object-oriented programming (OOP), sequence diagram, refactoring, Unified Modeling Language (UML)

I. Introduction

The evolution of software development leads to the necessity to detect common inefficient solutions for typical problems, called antipatterns. Who coined the term is still an arguable question. Some researchers [1, 2] consider Michael Akroyd [3] the creator of the term; Neill and Laplante [4] believe that Andrew Koenig [5] and Brown et al. [1] are worth mentioning. Identifying and eliminating antipatterns can significantly reduce the risks and improve the quality of the software produced. Clear definitions of antipatterns are required for correct identification.

Unified Modeling Language (UML) [6] is a popular modeling language used to describe a system visually. Visual descriptions of patterns and antipatterns allow repeatedly simplifying the study and understanding of the information. Furthermore, UML descriptions do not depend on the target programming language. However, the literature devoted to antipattern issues [1, 2, 7, 8] does not contain such descriptions, apart from design patterns literature [9]. We aim to develop UML diagrams for antipatterns in this paper.

In the literature [1], antipatterns are divided into the following categories: software development antipatterns, software architecture antipatterns, software project management antipatterns, and environmental antipatterns [4]. A programming paradigm can be also used as a criterion for further categorization. We focus on object-oriented programming (OOP) as the target paradigm. Antipatterns in OOP software development can be divided into design antipatterns and source code antipatterns. We selected design antipatterns for OOP as the object of our study.

UML descriptions for antipatterns can help to find bad solutions in a system that were included in the design stage. Clearly written description raises the level of the development and reduces the possibilities for errors in design and implementation.

In this paper, we propose our version of UML diagrams for OOP design antipatterns. Class diagrams are used to describe the structural aspects, whereas sequence diagrams are used to describe the behavioral aspects. Each antipattern description includes a refactoring approach. As an option, a design antipattern classification approach is offered.

Some of the antipatterns are known as an incorrect implementation of existing OOP patterns. This fact allows to discuss applicability of risky patterns to different systems and safety means for their implementation. Proposed refactoring methods make possible to test them in software development practices.

Related works are considered in the second section. A classification approach to antipatterns is presented in the third section. The fourth section describes the structural antipatterns and shows the corresponding UML diagrams. The fifth section lists behavioral antipatterns. The sixth section is devoted to creational antipatterns.

II. Related work

The lack of UML descriptions for antipatterns has already been studied. In "UML Specification and Correction of Object-Oriented Anti-patterns", Llano and Pooley [10] propose their own version of the UML diagrams for two antipatterns, God Object and Poltergeist. They used class diagrams to describe the structural features and state diagrams for behavior.

AntiPatterns: Refactoring Software, Architectures, and Project in Crisis [1] is important research in the area of code and design deficiencies that contains an extensive list of antipatterns with a detailed description of the symptoms and methods of solving problems. There are design problems, source code problems, project architecture problems, and project management problems. But this book does not contain UML diagrams of antipatterns.

Refactoring: Improving the Design of Existing Code [7] is a significant contribution to antipattern research. It contains a large list of code deficiencies, which the authors call the “code smell”, and detailed descriptions of the various refactoring methods. We assume that design defects discussed in the book such as Long Parameter List and Message Chains can also be attributed to antipatterns.

Antipattern detection in the source code has also drawn research attention. Din, AL-Badareen and Jusoh [2] provide an overview of literature on this issue, as well as a rich inventory of 22 existing antipatterns with brief descriptions. Additionally, one of the most recent works on antipattern detection [8] contains a fairly comprehensive literature review and detailed descriptions of antipatterns; however, fewer (11) antipatterns are named.

The above article by Llano and Pooley [10] contains diagrams of only two antipatterns, but diagrams of other antipatterns were not included. Thus, the problem of lack of UML diagrams for antipatterns has not yet been solved.

In our work, we propose UML diagrams for design antipatterns, as well as create the most extensive list of known antipatterns described in the literature to date.

III. Classifying design antipatterns

Recently, more than 20 antipatterns have been described [1, 2, 7, 8]. We propose to classify antipatterns based on the "goal" criterion: what problem the solution tries to solve, the same criterion as in [9]:

1) Structural antipatterns are antipatterns that determine the structural organization of objects and classes (for example, God Object, Sequential Coupling, and others). To describe such antipatterns with UML, we propose using a class diagram. For a more accurate description of some antipatterns (for example, Yo-yo Problem), a sequence diagram is added.

2) Behavioral antipatterns are antipatterns that determine the interaction between objects (for example, Poltergeists, Call Super, and others). To describe them, we propose using a class diagram and a sequence diagram.

3) Creational antipatterns are antipatterns that are an abstraction of the process of object creation. Singletonitis belongs to this group. We propose using class diagrams to describe solutions for this type.

IV. Structural design antipatterns

Structural antipatterns describe the unsuccessful use of inheritance and unsuccessful composition of classes and objects. This antipattern's subset may include the following:

A. Anemic Domain Model [11].

The domain class contains attributes and attribute access operations, rather than business logic operations. Martin Fowler defines the problem that arises when using this antipattern: "The fundamental horror of this anti-pattern is that it's so contrary to the basic idea of object-oriented design; which is to combine data and process together" [11].

This antipattern is quite controversial. For instance, in "The Anemic Domain Model is no Anti-Pattern, it's a SOLID design" [12], the author states:

If adherence to the SOLID principles is a property of well-engineered Object-Oriented programs, and an ADM adheres better to these principles than an RDM (Rich Domain Model), the ADM cannot be an antipattern, and should be considered a viable choice of architecture for domain modelling. [12]

We propose adding operations with business logic associated with this class. This modification should eliminate the problem described above.

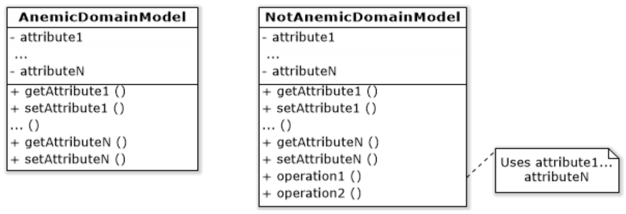

Fig. 1. Anemic Domain Model class diagram (left) and its refactoring (right)

B. God Object, (other name Blob) [1]

This class has a large number of attributes and operations that are not semantically related. It frequently interacts with Anemic Domain Model (God Object class contains business logic implementation associated with ADM classes). There are two forms, according to [1]:

1) God Object as a result of procedural design: a class with a large number of attributes and operations

This design eliminates all the advantages of OOP, because a large number of attributes and operations make it difficult to support and modify the system.

To resolve this problem, we can divide a class into multiple, more cohesive classes. This will simplify the understanding, support, and modification of the system fragment.

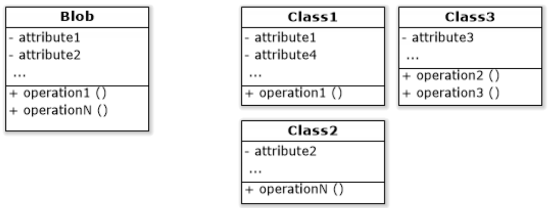

Fig. 2. God Object (B1) class diagram (left) and its refactoring (right).

2) God Object is associated with the so-called "data classes" (in fact, the data class is an antipattern, Anemic Domain Model)

In this case, the God Object acts as a processor class that processes the data. Such implementation does not correspond to the definition of the object as a collection of state and behaviour.

This implementation method is discussed by Llano and Pooley [10], who proposed the presentation of this antipattern on the state diagram.

We offer our version using a class diagram, because it is more appropriate to present the features of this architectural solution in a static diagram. This problem can be corrected by transferring methods that are semantically related to Anemic classes. This will both reduce the complexity of the God Object and correct the Anemic Domain models.

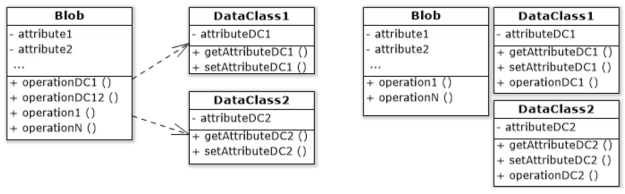

Fig. 3. God Object (B2) class diagram (left) and its refactoring (right)

C. *Constant Interface [2]*

This uses the interface as storage for constants only. Such use is contrary to the definition of the interface, which should describe the contract for objects implementing it. To correct the situation, it is necessary to add appropriate semantic operations to the interface.

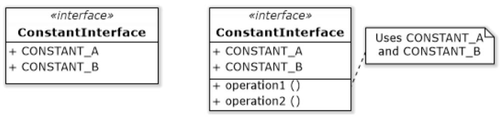

Fig. 4. Constant Interface class diagram (left) and its refactoring (right)

D. *Functional Decomposition [1]*

This is when a separate class is allocated to perform every new subtask in the system. It is the result of applying procedural design to OOP. Such design uses algorithms, rather than objects, as the main elements of constructing the system. This design contradicts the definition of OOP.

The method chosen for correcting this depends on the semantics. Fig. 5 presents a variant with the merger of all the subtask classes into one. This makes the object the main logical element of the system.

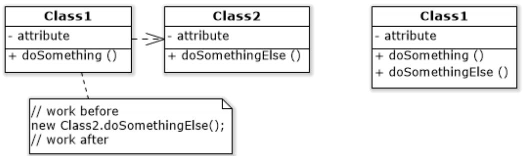

Fig. 5. Functional Decomposition class diagram (left) and its refactoring (right)

E. *Sequential Coupling [2]*

Sequential coupling refers to a class that requires its methods to be called in a particular sequence. This situation is the implementation of a certain behavior, the violation of which leads to errors that are difficult to detect. To eliminate the antipattern, we can combine methods from a sequence into one method.

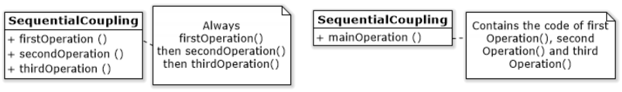
SequentialCoupling
+ firstOperation ()
+ secondOperation ()
+ thirdOperation ()
Always firstOperation() then secondOperation() then thirdOperation()
SequentialCoupling
+ mainOperation ()
Contains the code of first Operation(), second Operation() and third Operation()

Fig. 6. Sequential Coupling class diagram (left) and its refactoring (right)

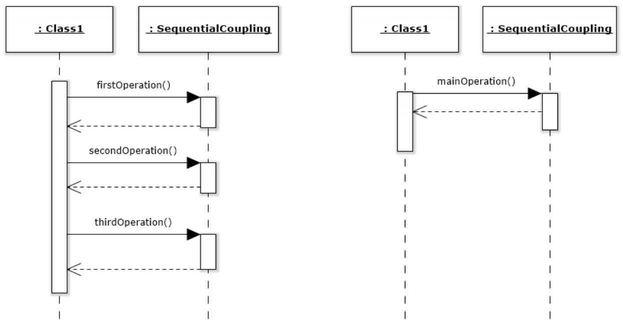

Fig. 7. Sequential Coupling sequence diagram (left) and its refactoring (right)

F. *Yo-yo Problem [2]*

Methods of different classes from the same hierarchy of inheritance are called in one control flow. This makes it very difficult to modify and search for errors. As a solution, changing the class hierarchy is suggested.

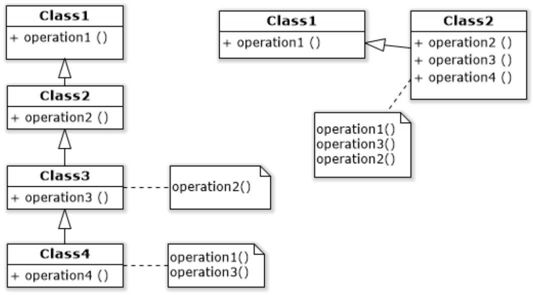

Fig. 8. Yo-yo Problem class diagram (left) and its refactoring (right)

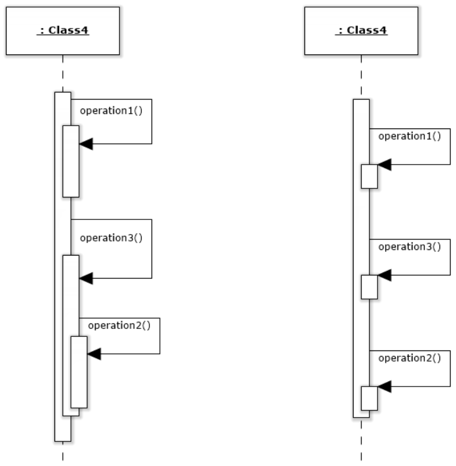

Fig. 9. Yo-yo Problem sequence diagram (left) and its refactoring (right)

G. *Swiss Army Knife [1]*

The class implements all possible use cases, which leads to a very complex interface. This antipattern is similar to the antipattern God Object (B1), so as an illustration, we can use a diagram (Fig. 2). The difference between these antipatterns is that the Swiss Knife class contains operations and attributes that will possibly be used, whereas all the attributes and operations of the God Object are used in the system. For refactoring, we can apply the method proposed for the God Object.

H. *Long Parameter List [7]*

This is a method that requires a large number of parameters. Fowler, Beck, Brant, Opdyke and Roberts [7] claim:

It is hard to understand such lists, which become contradictory and hard to use as they grow longer. Instead of a long list of parameters, a method can use the data of its own object. If the current object does not contain all necessary data, another object (which will get the necessary data) can be passed as a method parameter. (p. 65)

As a solution we can transfer the list of parameters to a separate class, but it can generate a new antipattern, Anemic Domain Model.

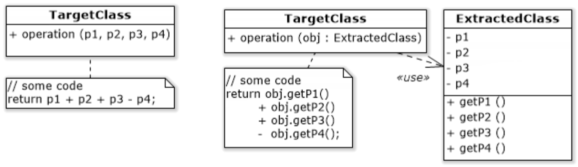

Fig. 10. Long Parameter List class diagram (left) and its refactoring (right)

I. *Refused Bequest [7]*

This class improperly implements inheritance. If the subclass, from the point of view of domain semantics, does not inherit the superclass and does not use part or all of the inherited functional methods, an antipattern Refused Bequest arises.

To eliminate the antipattern, we can transfer unused methods from the superclass. This structure of class hierarchy does not violate the principles of OOP.

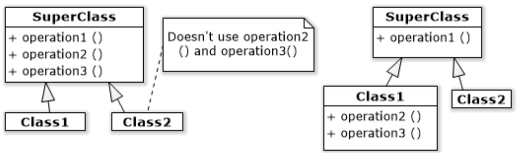

Fig. 11. Refused Bequest class diagram (left) and its refactoring (right)

J. *Shotgun Surgery [7]*

Changes in this class entail cascading changes in related classes. This situation significantly complicates the support and modification of the code, destroying all the benefits of using OOP.

To solve this problem, we need to eliminate redundant dependencies between classes, which will reduce the number of necessary changes. Fig. 12 presents an ideal version of refactoring, which eliminates all dependencies.

Fowler et al. [7] mention a particular case of this antipattern, Parallel Inheritance Hierarchies, which occurs when subclass creation of one class requires creating a subclass of another class.

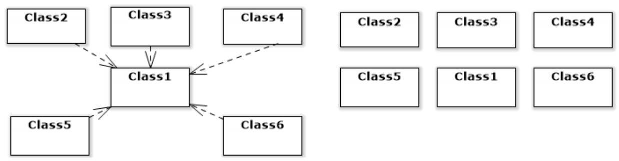

Fig. 12. Shotgun Surgery class diagram (left) and its refactoring (right)

V. Behavioral Design Antipatterns

A. *Base Bean [2]*

Base Bean is using inheritance to get access to functionality. This shortcoming violates the inheritance principle of OOP. To eliminate it, we suggest replacing the inheritance with dependence.

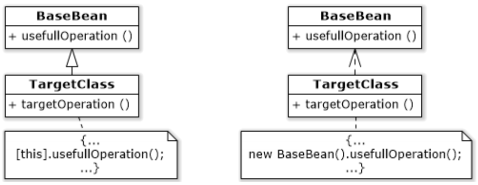

Fig. 13. Base Bean class diagram (left) and its refactoring (right)

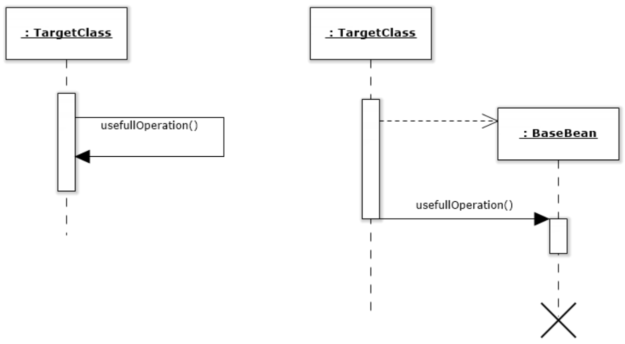

Fig. 14. Base Bean sequence diagram (left) and its refactoring (right)

B. *Call Super [2]*

Calling the method of the superclass is mandatory for overridden methods of subclasses. This greatly complicates the use and modification of those methods and can cause various errors. The solution to the problem can be the use of the Template Method pattern [9].

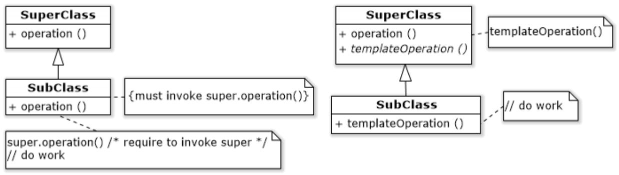

Fig. 15. Call Super class diagram (left) and its refactoring (right)

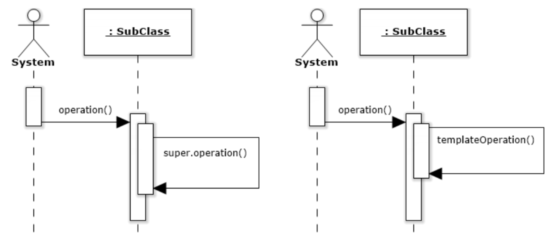

Fig. 16. Call Super sequence diagram (left) and its refactoring (right)

C. Poltergeist [1]

Poltergeist is a class with a very limited role and a short life cycle. This class is used as an auxiliary and does not reflect any essence of the business process, that is, it does not carry a state, which contradicts the principles of OOP. This antipattern is considered in detail by Llano and Pooley [10]. They offer five variants of antipattern implementation, refactoring methods, and corresponding activity and class diagrams. It should be added that the implementation called Irrelevant Classes essentially corresponds to the Anemic Domain Model antipattern.

D. Middle Man [7]

Using the delegation pattern [9] (transfer of part of the functional to the delegate class) where it is not necessary creates this antipattern. This complicates the understanding and support of code, and it can also generate classes that delegate absolutely all of their functionality to other classes, which contradicts the definition of the object of the system as a collection of state and behavior. The presence in each case of a pattern or antipattern is a problem of semantics that can only be solved by the developer.

Fowler et al. [7] proposed several refactoring methods to eliminate the antipattern. For example, if the role of the class delegator is negligible, we can eliminate the mediation class and access the delegate class directly. This structure is more obvious and amenable to modification.

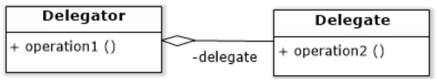

Fig. 17. Middle Man class diagram (Delegation pattern)

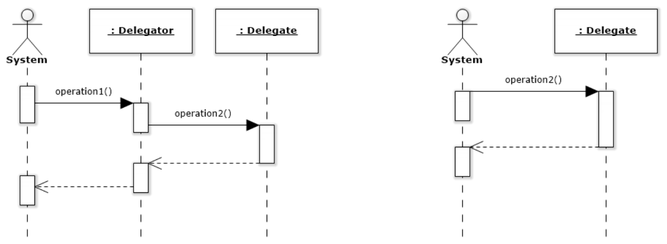

Fig. 18. Middle Man sequence diagram (left) and its refactoring (right).

E. Feature Envy [7]

This method makes a sequence of method calls of another class to get data and functionality. Because the application of the object-oriented approach implies the unification of the state and behaviour within a single object, it is necessary to perform refactoring. To correct the situation, we can transfer the logic to a class containing the state.

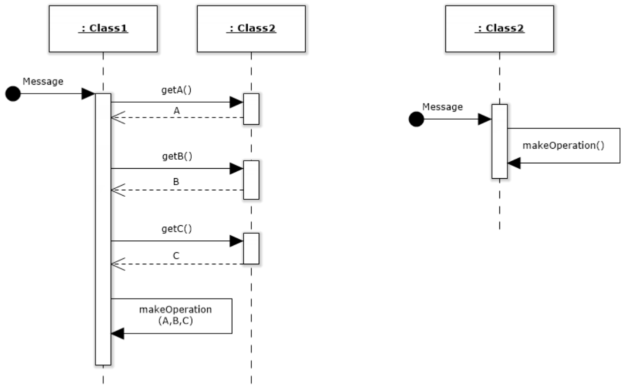

Fig. 19. Feature Envy sequence diagram (left) and its refactoring (right)

VI. Creational Design Antipatterns

A. Singletonitis [15]

Inappropriate use of the Singleton pattern leads to antipatterns. The creational pattern described in Design Patterns: Elements of Reusable Object-Oriented Software [9] has shortcomings. First, its implementation in many programming languages is not a guarantee of the uniqueness of the object created due to the peculiarities of implementing the work of parallel flows [15]. In addition, there are a number of difficulties in accessing a singleton object from parallel flows where its uniqueness is guaranteed.

From the point of view of OOP, the use of the Singleton creates a high coupling between parts of the Singleton class, which complicates the modification of the system. In the opinion of the author of “Singletonitis” [15], we should abstain from the use of the Singleton pattern as much as possible and apply it only where it is vital.

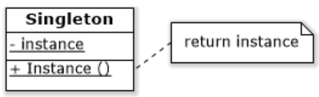

Fig. 20. Singletonitis class diagram (Singleton pattern)

B. Service Locator [16]

At the moment, there are disputes in the sphere of information technology about the nature of Service Locator: Fowler calls it a pattern [17], whereas Seemann [16] and McLean Hall [14] call it an antipattern. According to Seemann [16], the use of Service Locator makes it difficult to reuse the system and adds redundant dependencies (Class1 is dependent on ServiceLocator on Fig. 21).

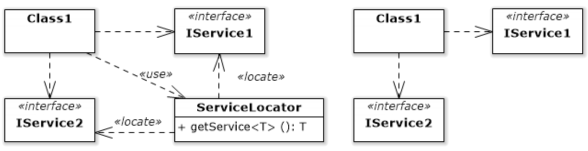

Fig. 21. Service Locator class diagram (left) and its refactoring (right)

To fix the problem, we can completely abandon the use of the Service Locator solution, or replace it with some other method of implementation of the Dependency Injection approach [16].

C. *Control Freak [16]*

This class has the “create” stereotype dependency. The presence in the system of such dependencies adversely affects the possibility of reuse and the testability of the system, and also contradicts one of the principles of OOP, low coupling [18].

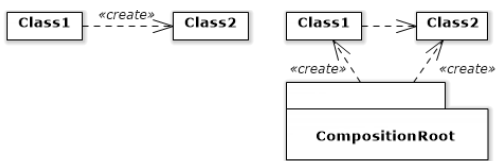

Fig. 22. Control Freak class diagram (left) and its refactoring (right)

To solve the problem, Seemann [16] proposes delegating the responsibilities for creating objects to a separate subsystem (or class), taking into account the peculiarities of the Dependency Injection approach. This solution will create dependencies between the factory subsystem and the other classes, but the coupling between the main classes of the system will weaken.

VII. Results and discussion

A deep examination of the antipatterns reviewed showed that the reason for some of them (Middle Man, Singletonitis, and Service Locator) is the incorrect use of design patterns (Delegation, Singleton, and Service Locator, respectively). The OOP principles violated by the antipattern are clearly indicated for each antipattern considered.

In addition to discussed above, the following antipatterns were analysed: Boat Anchor [1], Circular Dependency [2], Object Orgy [2], Circle-ellipse Problem [2], Interface Soup [14] and Message Chains [7]. But they were not included in this paper due to number of pages limit.

Representing antipatterns with UML not only greatly simplifies the process of assimilating new information in the study of antipatterns, but it can also serve as a guide for manually searching for antipatterns in class and sequence diagrams at the design stage of the system. Since UML is a universal representation language and does not impose restrictions on the choice of the system development language, UML representation of antipatterns is a universal tool for checking the correctness of the architecture of systems, including those already implemented.

Early flaws detection in the design stage of software development lifecycle will significantly reduce the cost of fixing flaws compared to detection during implementation or testing.

We believe that most of the described antipatterns cannot cause global problems in the developed system, however, they can negatively affect the cost of support and adding new features.

UML is an open standard and is constantly evolving, so we do not claim uniqueness for the proposed solutions. The results of the work will be applied to developing a new method for detecting design antipatterns on UML diagrams.

Acknowledgements

The authors like to thank Ms. Jean Kollantai and Ms. Lidia Artamonova at Tomsk State University for their valuable help in language editing.

References

[1] AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis / Brown, W.J., Malveau, R.C., McCormic III, H.W., Mowbray, T.J. , - New York: John Wiley & Sons, 1998.

[2] Din, J., AL-Badareen, A.B., Jusoh, Y.Y. Antipatterns detection approaches in Object-Oriented Design: A literature review. // Computing and Convergence Technology (ICCCT), 2012 7th International Conference on, Seoul, Rep. of Korea. 2012. - P. 926-931.

[3] Akroyd, M. AntiPatterns: Vaccinations against Object Misuse. - San Francisco: Object World West, 1996.

[4] Neill, C.J., Laplante, P.A. Antipatterns: Identification, Refactoring, and Management. - Boca Raton: CRC Press, 2005.

[5] Koenig, A. Patterns and Antipatterns // Journal of Object-Oriented Programming. - 1995. – Vol. 8, № 1. - P. 46-48.

[6] What is UML // Introduction to OMG’s Unified Modeling Language (UML) URL: http://www.uml.org/what-is-uml.htm (access date: 01.02.2021).

[7] Refactoring: Improving the Design of Existing Code / Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D. - Boston: Addison-Wesley Professional, 1999.

[8] Palomba, F., De Lucia, A., Bavota, G., Oliveto, R. Anti-Pattern Detection: Methods, Challenges, and Open Issues // Advances in Computers. - 2014. - №95. - P. 201-238.

[9] Design Patterns: Elements of Reusable Object-Oriented Software / Gamma, E., Helm, R., Johnson, R., Vlissides, J. - Boston: Addison-Wesley, 1995.

[10] Llano, M.T., Pooley, R. UML specification and correction of object-oriented anti-patterns // Proceedings of Fourth International Conference on Software Engineering Advances. - Los Alamitos: IEEE Computer Society, 2009. - P. 39-44.

[11] AnemicDomainModel // MartinFowler.com URL: http://www.martinfowler.com/bliki/AnemicDomainModel.html (access date: 01.02.2021).

[12] The Anaemic Domain Model is no Anti-Pattern, it’s a SOLID design // SAPM: Course Blog URL: https://blog.inf.ed.ac.uk/sapm/2014/02/04/the-anaemic-domain-model-is-no-anti-pattern-its-a-solid-design/ (access date: 01.02.2021).

[13] Martin R. Design principles and design patterns. // web.archive.org URL: https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf (access date: 01.02.2021).

[14] McLean Hall, G. Adaptive Code via C#: Agile coding with design patterns and SOLID principles. - Redmond: Microsoft Press, 2014.

[15] Singletonitis(2006) // Antonio's $HOME URL: http://www.antonioshome.net/blog/2006/20060906-1.php (access date: 01.02.2021).

[16] Seemann, M. Dependency Injection in .NET. - Greenwich: Manning Publications, 2011.

[17] Inversion of Control Containers and the Dependency Injection Pattern // MartinFowler.com URL: https://www.martinfowler.com/articles/injection.html (access date: 01.02.2021).

[18] Larman, C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development . - 3 edit. - Upper Saddle River: Prentice Hall, 2004.