МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2024
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2024
Proceedings of the Fifty-Third Spring Conference
of the Union of Bulgarian Mathematicians
Borovets, April 1–5, 2024

DESIGN PATTERNS OVER SOLID AND GRASP PRINCIPLES IN REAL PROJECTS*

Daniel Damyanov[1], Atanas Hristov[2], Zlatko Varbanov[3]
Department of Information Technologies,
"St. Cyril and St. Metodius" University of Veliko Tarnovo, Bulgaria
e-mails: [1]d.damyanov@ts.uni-vt.bg, [2]S1909012126@sd.uni-vt.bg,
[3]zl.varbanov@ts.uni-vt.bg

Design patterns (models) are repeatable and reusable constructions and solutions for common situations and problems to improve the quality of software systems. SOLID and GRASP are sets of design principles that are used in object-oriented software development. In this work we consider these solutions and principles and compare their (possibly combined) usage in different cases.
Keywords: design patterns, SOLID, GRASP, object-oriented design.

ШАБЛОНИ ЗА СОФТУЕРЕН ДИЗАЙН СРЕЩУ ПРИНЦИПИТЕ НА SOLID И GRASP В РЕАЛНИ ПРОЕКТИ

Даниел Дамянов[2], Атанас Христов[2], Златко Върбанов[3]
Катедра Информационни технологии,
Великотърновски университет „Св. св. Кирил и Методий", България
e-mails: [1]d.damyanov@ts.uni-vt.bg, [2]S1909012126@sd.uni-vt.bg,
[3]zl.varbanov@ts.uni-vt.bg

Шаблоните (моделите) за софтуерен дизайн са повтарящи се и многократно използвани конструкции и решения за често срещани ситуации и проблеми за подобряване на качеството на софтуерните системи. SOLID и GRASP са набори от принципи на проектиране, които се използват при разработка на обектно-ориентиран софтуер. В тази работа разглеждаме тези решения и принципи и сравняваме тяхното (евентуално комбинирано) използване в различни случаи.
Ключови думи: шаблони за софтуерен дизайн, SOLID, GRASP, обектно-ориентиран дизайн.

*https://doi.org/10.55630/mem.2024.53.076-084
2020 Mathematics Subject Classification: 68N19, 68U35.

1. Introduction. Much of the teaching and research literature argues that (GoF – "Gang of Four") patterns [2] increase the quality of object-oriented code, suggesting that design patterns greatly improve the quality of the systems where they are applied. Application requirements and functionalities grows up rapidly and the need for a dynamically continuous process of searching and developing new products or services is of utmost importance for companies to maintain a competitive advantage over their rivals [7]. Design patterns are repeatable and reusable constructions and solutions for common situations and problems faced by software engineers during development. The specific thing about them is that they are not language-specific and are easily implemented across platforms or domains, they simply alleviate the complexity associated with the design, or reduce the risk of trying to apply new functionality to the code [8]. Design patterns are also called a group of best practices for common design problems encountered by software engineers who use object-oriented software development. The main purpose when using patterns is to alleviate the complexity of creating and integrating objects into the program code. Once implemented, it is assumed that it will be easier to maintain the software project, as they act as a template that can be applied in real-life problems.

2. Design patterns, SOLID and GRASP.

2.1. Design patterns. Usually, whenever a problem occurs during development, you should consider implementing patterns to achieve resolution of the current problem, as they are risk-free, well researched and proven to be a reliable way to solve the specific problem[1]. In addition, they are used when the solution applied in the current project is outdated and the use of design patterns improves and maintains usability and readability. Design models are not limited to problem solving or simplifying issues, they are derived from structural engineering. Using design patterns can facilitate communication and collaboration between team members, and can facilitate integration as they are well known, common, and easy to recognize.

- Problems in using patterns: The use of design patterns can cause design problems to occur; It is well known that good software design has "high cohesion and low coupling". There is an empirical study trying to investigate whether the use of design patterns increases the modularity of the class or not[2]. The experiment was done by analyzing 5 open-source Java projects against the use of 17 design patterns, the results are finally shocking. The strict application of patterns results in classes using design patterns being highly related and less cohesive compared to classes that do not use design patterns.
- Incorrect use of design patterns: People tend to study and understand the design model without recognizing and learning when to apply it. However, some studies, e.g. [6, 9], suggest that the use of the design models does not always lead to "good" designs.
- Ineffective problem addressing: It is well known that the use of design patterns aims to adopt standardized best practice for a common problem. Yet the common problems are not the same, but there is at least one thing in common, which is most often overworking.

[1] https://betterprogramming.pub/4-things-to-consider-when-applying-design-patterns-46b9fcee4a59

[2] Empirical insight into the context of design patterns: Modularity analysis – IEEE Conference Publication (2017). [online] Available at: `http://ieeexplore.ieee.org/document/7549474/?part=1`

- Great similarity between design patterns: Design models have been criticized by many authors; it turns out that the 23 defined models in the GoF book [2] share something in common and do not provide any significant difference from other types of abstractions. Their argument is based on the MVC model, which is a form of structural abstraction that was defined earlier than when the GoF book was written. Of the 23 design models, Peter Norvig[3] proves that 16 of them are either simplified or eliminated in some dynamic languages such as LISP.
- Systematization of the most common mistakes in their use [1]:
  – increasing the complexity of the application;
  – over-engineering;
  – learning them;
  – code nonadaptability;
  – sensitivity to context;
  – lack of creativity;
  – increased indirectionality.

2.2. SOLID and GRASP. SOLID[4] is a popular set of design principles that are used in object-oriented software development. SOLID is an acronym that stands for five key design principles: single responsibility principle, open-closed principle, Liskov substitution principle, interface segregation principle and dependency inversion principle. All five are commonly used by software engineers and provide some important benefits for developers. The principles of SOLID were developed by Robert S. Martin in the essay "Design Principles and Design Models"[5] and were systematized and summarized in [5], although the acronym was later coined by Michael Feathers (see [5], p.58). In his essay, Martin acknowledges that successful software will change and evolve. As it changes, it becomes more complex. Martin warns that without good design principles, software becomes rigid, brittle, stationary and viscous. The principles of SOLID have been developed to combat these problematic design models. The overall objective of the SOLID principles is to reduce dependencies so that engineers modify one area of software without affecting others. By applying SOLID's principles, designers and developers can ensure that future updates are as painless as possible. The principles of SOLID are more supportive, scalable, tested and reusable.

The GRASP[6] (General Responsibility Assignment Software Principles) model include a set of principles and concepts that help in constructing applications using object-oriented programming. They list the responsibilities of classes and objects and are designed to make the code more flexible and extendable, making it easier to maintain [4]. GRASP is a set of exactly 9 software patterns for assigning general responsibility [3]. Assigning responsibilities to the site is one of the key skills of OOP. Every programmer and designer should be familiar with these models and, more importantly, know how to

[3]https://norvig.com/design-patterns/design-patterns.pdf

[4]Martin, Robert C. "Principles of OOD". ButUncleBob.com Archived from the original on Sep 10, 2014. Retrieved 2014-07-17

[5]Martin, Robert C. (2000),
`http://www.objectmentor.com/resources/articles/Principles\_and\_Patterns.pdf`

[6]Muhammad Umair (2018-02-26). "SOLID, GRASP, and Other Basic Principles of Object-Oriented Design"

apply them in their daily work (by the way, the same assumptions should apply to the principles of SOLID).

The main goal of software engineering is the development of appropriate software. SDLC (Systems development life cycle)[7] presents the steps of software engineering at the highest level. Software design is one of the steps of SLDC. Object-oriented software design is a popular approach of software design. To perform object-oriented software design there are many types of resources (in books, tutorials and documents) with different names such as: guidelines, principles, heuristics, models, styles, etc. The highest level is the basic principles, and the other levels are details that help to reach these basic principles. GoF are mentioned in Design Patterns. GRASP, in turn are design principles.

2.3. Design patterns vs. SOLID and GRASP. Design principles are general guidelines that can guide the class structure and the relationships between them. On the other hand, Design Patterns are proven solutions that solve frequently occurring problems. Most of the practical implementations of these design principles are primarily done using one or more design patterns. "Encapsulate what varies" considering fundamental design principles, this principle is contained in operation in almost all design models. If a component or module in the application has to be changed frequently, then it is good practice to separate that part of the code from the rest of the modules so that later on the part which varies can be extended or changed without affecting those which do not vary. Most design patterns such as Strategy, Adapter, Facade, Decorator, Observer, etc., follow this principle. At the same time, when using the patterns in large projects, especially at a stage when the application is working, implementing a pattern would be a real challenge, and usually not mandatory. The relationship between design patterns and SOLID principles is:

Design models and SOLID principles are not mutually exclusive. In fact, they often complement each other in software design. The SOLID principles provide guidance for designing individual classes and modules, while design patterns offer higher-level solutions for organizing classes and objects to solve common design problems.

SOLID principles help achieve the goals of modularity, extensibility and maintenance, while design models provide reusable solutions for more abstract design challenges. For example, the Strategy model can help with the Open-Closed (OCP) principle, as it allows easy change of algorithms or strategies without changing the existing codebase.

In practice, SOLID principles and design patterns can be used together to create well-designed, modular and flexible software systems.

2.4. Design analysis. In this section we look at real examples from the different approaches and make a comparative analysis. In order to reduce the volume of program code, we define the problem with short examples of code. Assume that a programmer has been given a task to create an application that must generate and change products and is saved in a log file for each operation performed. When considering the structure of the code, the programmer has started by saying that all products are actually separate components and each product must contain a list of its components, he will use the Composite pattern. Also, each item has a unique number that will be generated through the Singleton pattern. The product creation will use Factory Method to avoid tight cou-

[7] `https://ecampusontario.pressbooks.pub/informationsystemscdn/chapter/7-3-systems-development-life-cycle/`

pling in the implementation of services, and last but not least, to record every operation that happens, Observer will be used. For on-site storage purposes in the current article, we only show the UML diagram for the code to be executed in order to perform the corresponding operations (Figure 1):

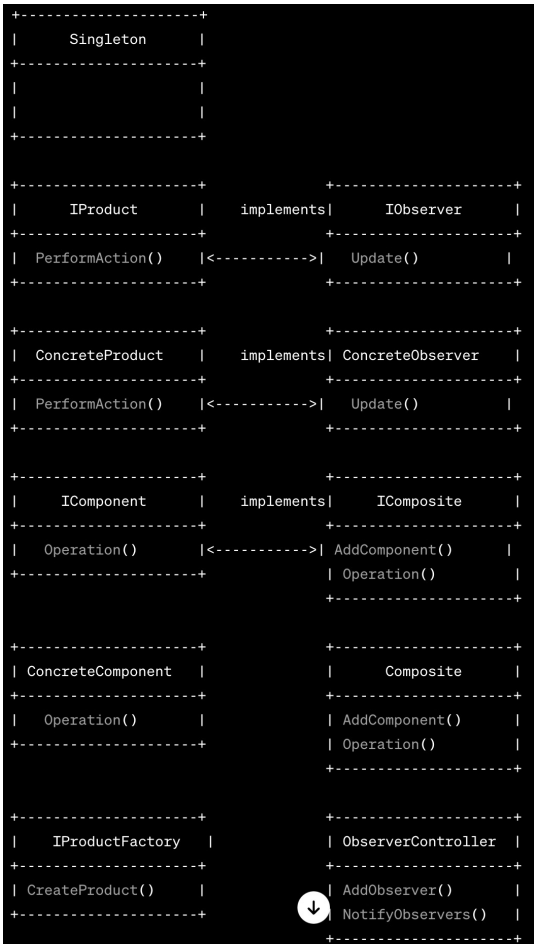

Fig. 1. Program implementing design patterns

The problem with this code is that different design patterns have been used without specific need and in a way that not only does not improve readability and code maintenance, but makes it extremely confused and difficult to understand. This example is indicative of how, with even a simple task, things can become too complicated right from the start. Instead of using design patterns to improve the structure and clarity of the code, they are added without meaning and create unnecessary complexity. This code is a good example of the fact that the use of design patterns should be considered and have a specific purpose. Cookies should not be added without taking into account the need for them and without striving for a clear and readable code. Otherwise, the code

becomes difficult to understand and maintain, losing the main meaning of using design paterns. Also, if you decide to change the code or add new functionality, it would be quite a complex task that will most likely take a long time. The probability that the code will cease to function according to the initial requirements is also not small. On the other hand, according to the same assignment, we can make the example only with the SOLID principles (Figure 2):

Fig. 2. UML of the application with SOLID principles

Changing the example shows several main advantages that are achieved when using the patterns:

Flexibility: For example, the Open/Closed Principle (OCP) promotes openness to expansion and closedness to change, making the system more flexible in adding new functionalities.

Maintainability: The Single Responsibility Principle (SRP) contributes to easier understanding and maintenance of classes, with each part of the system having clearly defined responsibility.

Testability: as the principles promote the separation of responsibilities and interfaces, which facilitates the isolation of components for testing.

Easy to Extend: principles such as OCP and LSP make the system more extensible

without requiring changes to existing code.

Low Coupling: SOLID promotes low inter-class dependence. This means that changes in one class will usually not necessitate changes in other classes, which reduces the risk of possible future problems.

Component Reusability: the creation of components that are easy to reuse is encouraged. This aspect facilitates the construction of more flexible and modular systems.

In the third example with GRASP the application has the following construction (Figure 3):

Fig. 3. GRASP UML diagram

The example shows a radically different picture than the design patterns. The structure is concentrated in the controller class, which takes care of the business logic of product operations. The creation of new instances is delegated to factors class that has the same purpose as that of the design patterns. The architecture of the application allows very easy addition of new dependencies, if necessary, respecting the "D" principle from SOLID. From the application made with GRASP principles, we can summarize several main advantages:

Simplification of design. This helps developers to get oriented and make design decisions easier.

Better maintenance. Objects that carry out their responsibilities clearly and effectively are easier to understand and maintain.

Code Reuse. Designing with GRASP can lead to fewer dependencies and less code-connected, making components more easily reusable. This can support the reuse of the code and speed up the development of new functionalities.

Greater flexibility. A well-defined code structure is encouraged, which makes the system more flexible for changes and addition of new functionalities. Principles such as "High Cohesion" and "Low Coupling" help to achieve this flexibility.

Easy testing. Code designed according to GRASP principles is usually easier to test. The individual components are well insulated, which facilitates the writing of tests and the execution of tests for each individual class or module.

Lower risk of errors. Common errors, such as excessive dependence or complex program logic, are avoided.

Also, we should note the shortcomings it entails, namely:

Lack of specifics. Some of the principles are more abstract, such as "Information Expert". This can lead to interpretations and difficulties in specifying these principles in concrete scenarios.

It does not deal with complex aspects. GRASP does not provide specific solutions for complex architectural problems such as parallelism, competitiveness or security.

Insufficient for larger projects. For larger and more complex projects that require more advanced architectural concepts and techniques, GRASP may proves to be insufficient. In these cases, other principles and practices such as SOLID principles, design templates and architectural patterns become more appropriate.

Doesn't solve all problems. GRASP is not a one-size-fits-all solution for all software design problems. It provides basic principles but does not solve all aspects of design.

Generally speaking, GRASP is a useful set of principles that can help in creating well-designed and easy-to-maintain software code. However, the best benefit is achieved when GRASP principles are combined with other software design principles and practices, such as SOLID, Design Patterns, and others. However, it is important to note that SOLID and templates are not opposing concepts, but are principles and methodologies for improving the quality and maintenance of software code. You can get a very good synergy between SOLID and Design Templates: SOLID principles and design templates often interact. For example, Strategy Pattern can help comply with the OCP principle. In turn, GRASP helps create readable, easy-to-understand code by assigning responsibilities to the appropriate classes. The principles of SOLID (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) and templates (such as GRASP – General Responsibility Assignment Software Patterns) are different approaches to software design. Instead of being mutually exclusive, they can be combined and used together to achieve better structure and support for software code. It is always desirable that programmers do not focus and implement absolutely everything in a given concept just to satisfy all its requirements, but to try to find the optimal option for a combination of several approaches that work in sync and achieve the initially set goals.

3. Conclusion. Skillful management of responsibilities in software development is the key to creating quality architecture and code. In combination with other models and practices, it is possible to develop well-crafted systems that are adaptable and easy to change. When approaching software design, it is useful to apply the SOLID and GRASP principles as a basis for creating well-structured classes and modules. These principles help to achieve good design practices and maintain code quality. However, design models can be used to address higher-level design challenges, such as managing complex interactions, separating components, or processing specific scenarios.

Using the principles of SOLID and GRASP ensures that individual classes adhere to good design practices. Then, when appropriate, design patterns can also be added to efficiently structure classes and objects and solve specific design challenges. It is worth noting that the application of design patterns should be guided by the specific requirements and context of the developed project. Not all design patterns are applicable in every situation, and it is important to take into account the compromises, complexity and suitability of the model before implementing them.

In summary, both SOLID/GRASP principles and design patterns play an important role in software design. SOLID/GRASP principles guide the design of individual classes and modules, while design models provide higher-level solutions to recurring design challenges. By understanding and applying both patterns and both approaches effectively, well-designed, modular, and flexible software systems can be developed.

REFERENCES

[1] W. Brown, R. Malveau, H. McCormick, T. Mowbray. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley Press, 1998.

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software", 1st edn. Addison-Wesley, 1994.

[3] C. Larman. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice Hall, 2001.

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

[5] R. Martin. Clean Architecture: A Craftsman's Guide to Software Structure and Design, 1st edn. Robert C. Martin Series, 2017

[6] W. McNatt, J. Bieman. Coupling of design patterns: Common practices and their benefits. In: Proceedings of the 25th Computer Software and Applications Conference, IEEE Computer Society Press, 2001, 574–579.

[7] T. Stefanov, S. Varbanova, M. Stefanova, I. Ivanov. CRM System as a Necessary Tool for Managing Commercial and Production Processes. *TEM Journal*, **12**, Issue 2 (2023), 785–797

[8] B. Venners. How to use design pattern. A Conversation with Erich Gamma, part I, May 2005 (available online at `https://www.immagic.com/eLibrary/archives/general/artimaus/A050628V.pdf`)

[9] P. Wendor. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In: Proceedings of 5th Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, 2001, 77–84.