УДК 004.4

В. А. Ковальський

ШАБЛОНИ ПРОЄКТУВАННЯ GRASP

Вінницький національний технічний університет

Анотація

Розглянуто та описано шаблони проектування GRASP, наведено приклади їх використання.
Ключові слова: шаблон, проєктування, програмне забезпечення.

Abstract

GRASP design templates are reviewed and described, and examples of their use are given.
Keywords: pattern, projecting, software.

Вступ

Важливою частиною інструментів розробника програмного забезпечення є використання шаблонів проєктування. Вони допомагають використовувати код повторно та надають загальні рішення типових проблем, які виникають під час проектування ПЗ. Використання шаблонів дозволяє писати більш зрозумілий код, зменшити кількість помилок та зробити принцип роботи ПЗ зрозумілішим. Важливим в парадигмі об'єктно-орієнтованого програмування є набір шаблонів GRASP.

Мета дослідження – розглянути та описати шаблони проектування GRASP.

Загальні відомості

GRASP(*General responsibility assignment software patterns* – загальні шаблони розподілу відповідальності) – набір шаблонів/принципів проєктування, які допомагають визначити обов'язки та взаємозв'язок об'єктів в проектованій системі.

Ці принципи допомагають розподілити обов'язки між класами в програмованій системі. Це дозволяє створювати більш зрозумілий код, зменшує його зв'язність та робить його модифікацію простішою. На відміну від класичних шаблонів проектування GoF шаблони GRASP не мають чітко окресленої структури та конкретних проблем, які вони вирішують. Вони є лише узагальненими принципами, які використовують при проектуванні програмного забезпечення[1].

Всього існує 9 таких принципів:

- Information Expert
- Creator
- Low Coupling
- High Cohesion
- Pure Fabrication
- Controller
- Polymorphism
- Indirection
- Protected Variations.

Опис принципів GRASP

Information expert(Інформаційний експерт) – принцип, який говорить про те, що за обробку інформації повинен бути відповідальним той, хто має її найбільше. Цей шаблон є інтуїтивно зрозумілим, адже трапляється і в реальному житті. До прикладу, філія деякого підприємства самостійно займається розподілом зарплат та премій відповідно до інформації про успіхи працівників, максимумом якої і володіє ця філія.

Приклад використання даного шаблону на мові Java зображений на рисунку 1. Клас компанія містить список всіх робітників, а отже розраховує суму всіх заробітних плат.

Рисунок 1 – Принцип «Інформаційний експерт»

Creator(Творець) – принцип, який наділяє клас відповідальністю створення інших об'єктів за певних умов, де клас-творець:

- агрегує або містить об'єкти, які створює
- використовує створювані об'єкти
- має дані для їх ініціалізації.

Приклад реалізації даного шаблону проєктування на мові Java зображений на рисунку 2. Компанія в прикладі містить список робітників, інформацію необхідну для їх найму і може використовувати їх, а отже до класу Company можна і варто використати шаблон «Творець». Створення об'єктів(робітників компанії) реалізовано в методі hireEmployee[1].

```
class Company {

    private final List<Employee> employees = new LinkedList<>();

    public void hireEmployee(String name, int salary) {
        if(employees.size() < 50) {
            Employee newEmployee = new Employee(name, salary);
            employees.add(newEmployee);
        }
    }
}
```

Рисунок 2 – Шаблон «Творець»

Альтернативним до цього шаблону є шаблон «Абстрактна фабрика», який пропонує розробку окремого класу, який наділений відповідальністю створення об'єктів іншого[2].

Low Coupling(Слабка зв'язність) – шаблон, який пропонує зменшити зв'язність між класами у коді. Сильна зв'язність об'єктів спричиняє багато проблем, ускладнює модифікацію коду, а зміна одного об'єкту може вплинути на всі інші, які з ним пов'язані. Для зменшення зв'язності рекомендується використання інтерфейсів з їх реалізацією замість конкретних реалізацій. В такому випадку за необхідності код можна буде легко змінити змінюючи різні реалізації одного й того самого інтерфейсу і зв'язність коду буде зменшена[1]. Аналогом цього шаблону в принципах SOLID є The Dependency Inversion Principle[3].

Як приклад успішного використання інтерфейсів для зменшення зв'язності коду можна взяти вбудовану бібліотеку Java Collections API. Ця бібліотека містить інтерфейс List і декілька його реалізацій, тому під час кодування рекомендується створювати змінну типу List, після чого вона зможе посилатись на конкретні реалізації. При цьому за необхідності реалізації можна замінити прямо в коді. В прикладі нижче спочатку створюється список типу ArrayList, потім змінюється на тип LinkedList зі

збереженням вмісту, після аналогічно змінюється на тип CopyOnWriteArrayList[4]. Використання інтерфейсу List зображено на рисунку 3.

```
public static void main(String[] args) {

    List<String> list = new ArrayList<>();

    list = new LinkedList<>(list);

    list = new CopyOnWriteArrayList<>(list);
}
```

Рисунок 3 – Використання інтерфейсів для зменшення зв’язності

High Cohesion(Висока зчепленість) – шаблон, який говорить про те, що клас повинен виконувати якнайменшу кількість неспецифічних для нього функцій і мати певну конкретну область використання[1]. Цей принцип є по суті є аналогом одного з принципів SOLID Single Responsibility Principle, який говорить про те, що кожний клас повинен виконувати одне завдання[3].

Існує думка про необхідність балансування в коді між Low Coupling та High Cohesion, адже неправильне використання першого шаблону може призвести до проблем з другим і навпаки.

Pure Fabrication(Повна вигадка) – шаблон, який пропонує використання службових класів, аналоги яких не існують в предметній області ПЗ[1].

До прикладу, зазвичай для зв’язку з базою даних у веб-додатках створюється окремий клас-сервіс. Очевидно, що в предметній області немає реальної сутності, яка відповідала б такому класу. У випадку якщо наділити класи реальних сутностей функціональністю такого службового класу, буде порушений принцип високої зчепленості за низької зв’язності: класу доведеться виконувати додаткові функції і зв’язуватись з базою даних, що зробить код менш читабельним та більш комплексним.

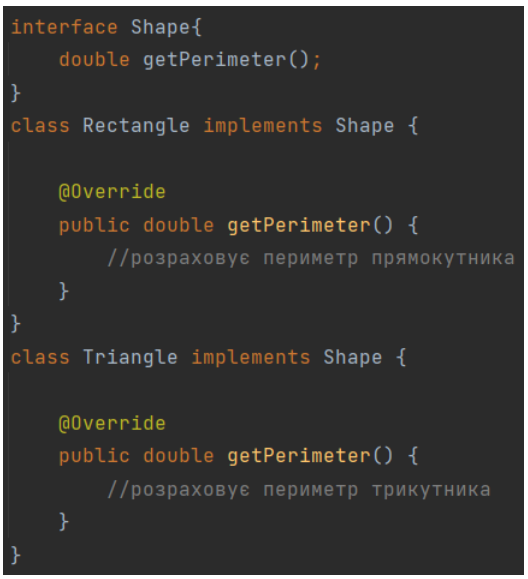
Звісно, створення додаткових класів дещо ускладнює систему, проте цей шаблон робить принцип її роботи зрозумілішим.

Controller(Контролер) – шаблон, який має за мету розділити інтерфейс(представлення інформації) додатку від його логіки. Він займається обробкою всіх вхідних запитів і делегує їх між іншими класами.

Популярним цей принцип є при розробці веб-додатків, які працюють за парадигмою MVC(Model Controller View).

Polymorphism(Поліморфізм) – один з принципів GRASP, який водночас є одним з фундаментальних принципів об’єктно-орієнтованого програмування. Цей шаблон пропонує альтернативні варіанти поведінки на основі типу об’єктів, які реалізують один і той самий інтерфейс.

Як приклад використання поліморфізму можна навести класи Rectangle та Triangle, які реалізують інтерфейс GeometryShape. Об’єкти цих класів можуть викликати метод getPerimeter. При цьому периметр фігури розраховується відповідно до класу об’єкту, тобто виконується метод, який описаний у відповідному класі. Приклад поліморфізму на мові Java зображений на рисунку 4.



```
interface Shape{
    double getPerimeter();
}
class Rectangle implements Shape {

    @Override
    public double getPerimeter() {
        //розраховує периметр прямокутника
    }
}
class Triangle implements Shape {

    @Override
    public double getPerimeter() {
        //розраховує периметр трикутника
    }
}
```

Рисунок 4 – Приклад використання поліморфізму

Indirection(Перенаправлення або посередник) – принцип, який полягає у використанні проміжного класу для створення зв'язку між певним об'єктом та іншим об'єктом, функціонал якого він потребує. Тобто взаємодія між класами зосереджується в окремому класі. Таким чином зміншується зв'язність коду, проте погіршується його читабельність. Даний принцип відповідає GoF шаблону «Медіатор»[2]. Варто зауважити, що шар Controller в моделі MVC є посередником між моделлю додатку та відображенням.

Можна повернутись до прикладу взаємозв'язку між веб-додатком та базою даних, де створюється службовий клас, який є посередником між ними.

Protected Variations(Стійкість до змін). Під час проєктування необхідно розробити систему таким чином, щоб компоненти були стабільними та піддавались майбутній модифікації без поганого впливу на інші компоненти. Цю проблему вирішує шаблон «Стійкість до змін». За цим шаблоном необхідно визначити можливі «точки нестабільності» створити для них інтерфейси і реалізувати різні варіанти поведінки.

Наприклад, існує компанія з продажу паперу та прогнозується її розширення, після якого вона буде займатися ще й продажом принтерів. В такому випадку варто створити інтерфейс Product, який реалізують класи Paper та Printer.

В певному сенсі принцип «Стійкість до змін» узагальнює всі принципи GRASP, адже всі вони є по суті рекомендацією для успішного проектування стійких систем, в яких існує баланс між низькою зв'язністю та високою зчепленністю.

Висновки

Всі 9 принципів GRASP між собою пов'язані і деякі з них підкріплюють інші. Хоч вони і не вирішують певні конкретні проблеми і не схожі на класичні шаблони GoF, їх використання є важливим при проєктуванні архітектури системи. Правильне їх розуміння дозволяє створювати більш читабельний код та спроектувати систему із зрозумілим принципом роботи та стабільними компонентами.

Отже, було описано всі 9 принципів GRASP та наведено приклади їх використання.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. GRASP принципы[Електронний ресурс] – Режим доступу: https://bool.dev/blog/detail/grasp-printsipy(дата звернення 01.03.2023).
2. Патерни проектування[Електронний ресурс] – Режим доступу: https://refactoring.guru/uk/design-patterns(дата звернення 01.03.2023).
3. Чому SOLID – важлива складова мислення програміста[Електронний ресурс] – Режим доступу: https://dou.ua/lenta/articles/solid-principles/(дата звернення 03.03.2023).

4. Interface List<E>[Електронний ресурс] – Режим доступу: https://docs.oracle.com/javase/8/docs/api/java/util/List.html(дата звернення 06.03.2023).

Ковальський Валентин Анатолійович, студент групи ЗПІ-21б, Факультет інформаційних технологій та комп'ютерної інженерії.