Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №4
Технологія розроблення програмного забезпечення

Тема: «ШАБЛОНИ «SINGLETON», «ITERATOR», «PROXY», «STATE», «STRATEGY»

Варіант 25

Виконала
студентка групи ІА-24
Мелешко Юлія Сергіївна

Перевірив:
Мягкий Михайло
Юрійович

Київ 2024р.

Мета: Навчитися використовувати шаблони singleton, iterator, proxy, state, strategy

Завдання. 1. Ознайомитися з короткими теоретичними відомостями.

2. Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

3. Застосування одного з розглянутих шаблонів при реалізації програми

..25 Installer generator (iterator, builder, factory method, bridge, interpreter, client-server)

Генератор інсталяційних пакетів повинен мати якийсь спосіб налаштування файлів, що входять в установку, установки вікон з інтерактивними можливостями (галочка - створити ярлик на робочому столі; ввести в текстове поле деякі дані, наприклад, ліцензійний ключ і т.д.). Генератор повинен вивести один файл .exe або .msi.

Код: https://github.com/jlmp3mp44/trpz/tree/lab3

Теоретичні відомості

Патерни проектування

Патерн проектування — це типовий спосіб вирішення певної проблеми, що часто

зустрічається при проектуванні архітектури програм. На відміну від готових функцій чи бібліотек, патерн не можна просто взяти й скопіювати в програму. Патерн являє собою не якийсь конкретний код, а загальний принцип вирішення певної проблеми, який майже завжди треба підлаштовувати для потреб тієї чи іншої програми. Патерни часто плутають з алгоритмами, адже обидва поняття описують типові рішення відомих проблем. Але якщо алгоритм — це чіткий набір дій, то патерн — це високорівневий опис рішення, реалізація якого може відрізнятися у двох різних програмах. Якщо провести аналогії, то алгоритм — це кулінарний рецепт з чіткими кроками, а патерн — інженерне креслення, на якому намальовано рішення без конкретних кроків його отримання. Найбільш низькорівневі та прості патерни — ідіоми. Вони не дуже універсальні, позаяк мають сенс лише в рамках однієї мови програмування.

Найбільш універсальні — архітектурні патерни, які можна реалізувати практично

будь-якою мовою. Вони потрібні для проектування всієї програми, а не окремих її

елементів. Крім цього, патерни відрізняються і за призначенням. Існують три основні групи патернів:

• Породжуючі патерни піклуються про гнучке створення об’єктів без внесення в

програму зайвих залежностей.

• Структурні патерни показують різні способи побудови зв’язків між об’єктами.

• Поведінкові патерни піклуються про ефективну комунікацію між об’єктами.

Одинак (Singleton)

Шаблон проєктування "Одинак" гарантує, що клас матиме лише один екземпляр, і забезпечує глобальну точку доступу до цього екземпляра. Це корисно, коли потрібно контролювати доступ до деяких спільних ресурсів, наприклад, підключення до бази даних або конфігураційного файлу. Основна ідея полягає у тому, щоб закрити доступ до конструктора класу і створити статичний метод, що повертає єдиний екземпляр цього класу. У мовах, які підтримують багатопоточність, також важливо синхронізувати метод доступу, щоб уникнути створення кількох екземплярів в різних потоках. Один із способів реалізації одинак у Java – використання статичної ініціалізації, яка автоматично забезпечує безпечність у багатопоточному середовищі.

Деякі розробники вважають одинак антипатерном, оскільки він створює глобальний стан програми, що ускладнює тестування та підтримку. Використання цього шаблону має бути обґрунтованим і обмеженим певними обставинами.

Ітератор (Iterator)

Шаблон "Ітератор" надає спосіб послідовного доступу до елементів колекції без розкриття її внутрішньої структури. Він особливо корисний для обходу різних типів колекцій, таких як списки, множини або деревовидні структури, незалежно від того, як вони реалізовані. Ітератор інкапсулює поточний стан перебору, тому може зберігати інформацію про те, який елемент є наступним. Цей шаблон дозволяє відокремити логіку роботи з колекцією від логіки обходу, що робить код більш гнучким і модульним. У Java цей шаблон реалізований у вигляді інтерфейсу Iterator, який надає методи hasNext() та next() для послідовного перебору елементів. Ітератор також можна використовувати для видалення елементів під час обходу колекції. Завдяки цьому шаблону можна використовувати поліморфізм для однакового доступу до елементів різних колекцій.

Проксі (Proxy)

Шаблон "Проксі" створює замісник або посередника для іншого об’єкта, що контролює доступ до цього об’єкта. Проксі може виконувати додаткову роботу перед передачею викликів реальному об'єкту, як-от перевірку прав доступу або відкладену ініціалізацію. Існує декілька типів проксі, серед яких захисний проксі, який контролює доступ, і віртуальний проксі, який затримує створення об'єкта, поки він не буде потрібен. У Java цей шаблон часто використовується для створення динамічних проксі за допомогою інтерфейсів, де проксі-клас реалізує той самий інтерфейс, що й реальний об’єкт. Проксі ефективний для оптимізації

роботи з ресурсами або для контролю доступу до важких у створенні об'єктів. Це дозволяє зберігати оригінальний об'єкт захищеним і надає додатковий шар для маніпуляцій.

Стан (State)

Шаблон "Стан" дозволяє об'єкту змінювати свою поведінку при зміні внутрішнього стану, надаючи йому різні стани для різних контекстів. Він ефективно інкапсулює різні стани об'єкта як окремі класи і делегує дії поточному стану. Наприклад, кнопка може мати різні дії залежно від того, чи вона активована чи деактивована. Це дозволяє замість довгих умовних операторів використовувати поліморфізм, де кожен клас стану реалізує свою поведінку, визначену інтерфейсом стану. У Java цей шаблон може бути реалізований як клас з інтерфейсом для станів, де кожен стан є окремим підкласом, що відповідає за певну поведінку. Це полегшує масштабування та тестування, оскільки додавання нового стану не вимагає змін у вихідному коді.

Стратегія (Strategy)

Шаблон "Стратегія" дозволяє вибирати алгоритм або поведінку під час виконання, забезпечуючи взаємозамінність різних алгоритмів для конкретного завдання. Він передбачає інкапсуляцію різних варіантів поведінки в окремих класах, які реалізують один інтерфейс, що спрощує заміну і додавання алгоритмів. Клас контексту отримує об'єкт стратегії і викликає відповідні методи, не знаючи деталей реалізації конкретної стратегії. Наприклад, клас сортування може мати кілька стратегій: швидке сортування, сортування вставкою чи сортування вибором, і залежно від контексту обирається відповідний метод. У Java шаблон реалізується через інтерфейс стратегії, який мають різні класи конкретних стратегій.

Хід роботи

Я вирішила використатти патерн «State»

Патерн "Стан" гарно відповідає моїм вимогам, оскільки дозволяє організувати етапи як окремі, логічно незалежні стани. Кожен стан гарантує правильний порядок виконання дій, запобігаючи таким некоректним операціям, як спроба створення інсталятора до додавання файлів.

Ключовою перевагою патерна є його здатність чітко відокремлювати поведінку кожного етапу. Наприклад, у **початковому стані** користувач може лише розпочати роботу, у **стані вибору файлів** — додавати файли, а у **стані конфігурації** — налаштовувати параметри, такі як створення ярликів або введення ліцензійного ключа. Лише у **стані готовності** стає доступною функція

генерації інсталятора. Це розділення робить структуру коду зрозумілою, спрощуючи його підтримку та розширення.

Інтерактивний процес створення інсталятора є ще одним важливим аспектом мого проєкту. Патерн "Стан" дозволяє кожному стану самостійно управляти своїми вимогами, такими як створення ярликів або налаштування конфігурацій. Це забезпечує плавний і зрозумілий досвід для користувача, чітко вказуючи, на якому етапі він знаходиться та які дії доступні в даний момент.

Крім того, патерн "Стан" вбудовує валідацію безпосередньо у свою структуру. Це означає, що користувач не зможе виконувати некоректні дії, наприклад, налаштовувати параметри до вибору файлів або генерувати інсталятор без завершення налаштувань. Кожен стан має власні правила валідації, що сприяє цілісності даних і правильності роботи програми.

Окрім технічних переваг, патерн "Стан" покращує досвід користувача. Користувачі отримують чіткий зворотний зв’язок на кожному етапі, що дозволяє їм виконувати лише валідні дії.

Таким чином, патерн "Стан" є ідеальним вибором для мого додатку генерації інсталяторів. Він забезпечує не лише надійну реалізацію послідовного процесу, але й сприяє підтримці зрозумілого, масштабованого та легко адаптованого коду.

Структура шаблону «State»

```
public interface InstallerState {
    4 implementations
    String initializeInstaller(User user);
    4 implementations
    boolean addFiles(List<File> files);
    4 implementations
    void configure(String shortcutOption, String licenseKey, List<String> options);
    4 implementations
    String generateInstaller();
    4 implementations
    String getStateName();
}
```

Імплементації інтерфейсу

```java
public class ConfigurationState implements InstallerState {
    private final InstallerGenerator installer;

    Julya
    public ConfigurationState(InstallerGenerator installer) { this.installer = installer; }

    Julya
    @Override
    public String initializeInstaller(User user) { return "Installer already initialized"; }

    Julya
    @Override
    public boolean addFiles(List<File> files) { return false; // Not allowed in this state }

    Julya
    @Override
    public void configure(String shortcutOption, String licenseKey, List<String> options) {
        installer.setShortcut(shortcutOption);
        installer.setInstallationOption(String.join( delimiter: ",", options));
        installer.setState(new ReadyState(installer));
    }

    Julya
    @Override
    public String generateInstaller() { return "Cannot generate installer before configuration"

    Julya
    @Override
    public String getStateName() { return "Configuration State"; }
}
```

```java
public class FileSelectionState implements InstallerState {
    private final InstallerGenerator installer;

    Julya
    public FileSelectionState(InstallerGenerator installer) { this.installer = installer; }

    Julya
    @Override
    public String initializeInstaller(User user) { return "Installer already initialized"; }

    Julya
    @Override
    public boolean addFiles(List<File> files) {
        try {
            for (File file : files) {
                installer.setFile(file); // Note: This might need to be modified to handle multiple fi
            }
            installer.setState(new ConfigurationState(installer));
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    Julya
    @Override
    public void configure(String shortcutOption, String licenseKey, List<String> options) {
        // Not allowed in this state
    }

    Julya
    @Override
    public String generateInstaller() { return "Cannot generate installer without configuration"; }

    Julya
    @Override
    public String getStateName() { return "File Selection State"; }
}
```

```java
Julya
public class InitialState implements InstallerState {
    private final InstallerGenerator installer;

    Julya
    public InitialState(InstallerGenerator installer) { this.installer = installer; }

    Julya
    @Override
    public String initializeInstaller(User user) {
        installer.setUser(user);
        installer.setState(new FileSelectionState(installer));
        return "Installer initialized, ready for file selection";
    }

    Julya
    @Override
    public boolean addFiles(List<File> files) { return false; // Not allowed in this state }

    Julya
    @Override
    public void configure(String shortcutOption, String licenseKey, List<String> options) {
        // Not allowed in this state
    }

    Julya
    @Override
    public String generateInstaller() { return "Cannot generate installer in initial state"; }

    Julya
    @Override
    public String getStateName() { return "Initial State"; }
}
```

```java
public class ReadyState implements InstallerState {
    private final InstallerGenerator installer;


    public ReadyState(InstallerGenerator installer) { this.installer = installer; }


    @Override
    public String initializeInstaller(User user) { return "Installer already initialized"; }


    @Override
    public boolean addFiles(List<File> files) { return false; // Not allowed in this state }


    @Override
    public void configure(String shortcutOption, String licenseKey, List<String> options) {
        // Can reconfigure if needed
        installer.setShortcut(shortcutOption);
        installer.setInstallationOption(String.join( delimiter: ",", options));
    }


    @Override
    public String generateInstaller() {
        // Here would be the actual logic to generate the installer
        return "Generating installer with configuration: " + installer.getInstallationOption();
    }


    @Override
    public String getStateName() { return "Ready State"; }
}
```

Висновок: Під час лабораторної роботи я обрала підходящий шаблон, та реалізувала його, Я обрала шаблон State, що буде контролювати дії користувача відповідно до етапів. Також було реалізовано декілька сервісів для взаємодії класів.