Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Лабораторна робота №7

Технологія розробки програмного забезпечення

Шаблони «Mediator», «Facade», «Bridge», «Template Method»

Виконала студентка
групи ІА-23:
Кашуб’як С. М.

Перевірив:
Мягкий М. Ю.

Київ 2024

Тема: Шаблони «Mediator», «Facade», «Bridge», «Template Method».

Мета: Ознайомитися з принципами роботи шаблонів проектування «Mediator», «Facade», «Bridge», «Template Method», їх перевагами та недоліками. Набути практичних навичок у застосуванні шаблону adapter при розробці програмного забезпечення на прикладі реалізації архіватора.

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

Варіант:

..14 Архіватор (strategy, adapter, factory method, facade, visitor, p2p)

Архіватор повинен являти собою візуальний додаток з можливістю створення і редагування архівів різного типу (.tar.gz, .zip, .rar, .ace) додавання/ видалення файлів / папок, редагування метаданих (по можливості), перевірка checksum архівів, тестування архівів на наявність пошкоджень, розбиття архівів на частини.

ЗМІСТ

Теоретичні відомості

Принцип "Don't Repeat Yourself" (DRY) – це програмістський принцип, який закликає уникати повторення одного й того ж коду або інформації в програмному забезпеченні. Ідея полягає в тому, щоб написати код один раз і використовувати його в багатьох місцях, замість того щоб дублювати один і той же код у різних частинах програми.

Основні положення DRY:

1. Уникнення дублювання коду: Якщо одна і та ж інформація або функціональність міститься в декількох місцях програми, її слід об'єднати в одному місці. Це підвищує підтримуваність і знижує ймовірність помилок, пов'язаних з оновленням кодової бази.
2. Ефективне повторне використання коду: Завдяки DRY, коди можуть бути повторно використані без внесення змін у декілька місць, що робить їх менш вразливими до помилок і спрощує внесення змін.
3. Краща організація коду: Дотримання DRY допомагає організувати кодову базу більш чітко і логічно, що сприяє кращому розумінню програмного коду та полегшує процес розробки і підтримки програмного забезпечення.

Шаблон Фасад (Facade) – структурний шаблон проектування, який надає простий, єдиний інтерфейс до складної підсистеми об'єктів. Фасад приховує складність підсистеми за допомогою одного універсального інтерфейсу, що дозволяє клієнтам взаємодіяти з нею без необхідності знати її внутрішню структуру або реалізацію.

Як працює Фасад:

1. Facade (Фасад) – це клас, який забезпечує простий інтерфейс для доступу до складної підсистеми. Фасад виконує роль інтермедіатора, приховуючи деталі функціонування підсистеми та надаючи єдину точку входу для взаємодії з нею.
2. Subsystems (Підсистеми) – це складні компоненти системи, що обробляють різні аспекти функціональності. Кожна підсистема має свій власний інтерфейс і функціональність.
3. Client (Клієнт) – це об'єкт, який взаємодіє з фасадом, використовуючи простий інтерфейс, не звертаючись безпосередньо до підсистем.

Переваги:

- Спрощення інтерфейсу – клієнтам не потрібно мати справу з деталями підсистеми, що робить код більш зрозумілим і легким у підтримці.
- Ізоляція змін – зміни, що відбуваються у підсистемі, не впливають на фасад або клієнтів, що дозволяє змінювати підсистему без необхідності коригування коду фасаду або клієнта.

- Гнучкість – нові компоненти або підсистеми можуть бути легко додані до фасаду без зміни існуючих клієнтів.

Недоліки:

- Занадто багато обов’язків - фасад може стати занадто великим і складним, якщо він намагається приховати надто багато деталей, що знижує його ефективність.
- Обмеження доступу - фасад обмежує прямий доступ до підсистеми, що може бути недоцільним у деяких випадках, коли потрібен прямий доступ для спеціальних операцій.

Хід Роботи

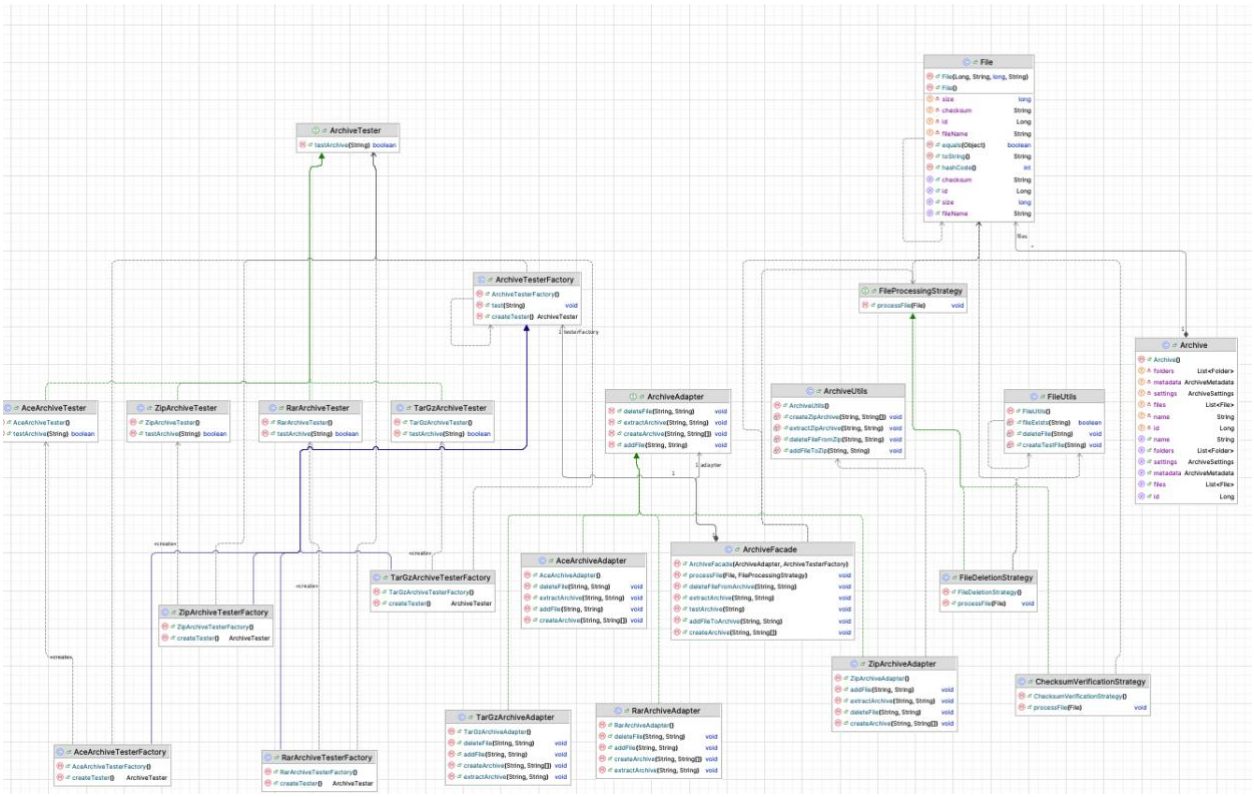

Рис 1. Діаграма класів

Діаграма класів

Діаграма демонструє реалізацію шаблону "Facade" у проєкті архіватора, який спрощує доступ до основних операцій із архівами, приховуючи складність внутрішніх компонентів системи. ArchiveFacade є центральним класом, що забезпечує взаємодію між клієнтським кодом та підсистемами архівування, тестування та обробки файлів.

ArchiveFacade – це головний клас, який надає клієнтам єдиний інтерфейс для роботи з архівами.

Основні методи:

- createArchive – створює архів із вказаних файлів.
- extractArchive – розпаковує архів у задану директорію.
- addFileToArchive – додає файл до існуючого архіву.
- deleteFileFromArchive – видаляє файл із архіву.
- testArchive – перевіряє цілісність архіву.
- processFile – застосовує стратегію для обробки файлів.

Adapter Pattern - фасад взаємодіє з архівами через інтерфейс `ArchiveAdapter`, що дозволяє підключати різні типи архівів. `ArchiveAdapter` – інтерфейс для роботи з архівами. Перевага: завдяки цьому інтерфейсу ArchiveFacade може працювати з будь-яким типом архіву без зміни власного коду.

Для перевірки архівів фасад використовує фабрику тестувальників. `ArchiveTesterFactory` – абстрактний клас, що створює об'єкти для перевірки архівів.

Тестувальники архівів реалізують інтерфейс `ArchiveTester`, який містить метод testArchive. Це забезпечує єдиний підхід до тестування архівів різних форматів.

Фасад також підтримує стратегії обробки файлів, дозволяючи застосовувати різні операції до файлів. `FileProcessingStrategy` – інтерфейс для стратегій обробки файлів.

Клієнтський код взаємодіє виключно з ArchiveFacade, викликаючи методи для роботи з архівами.

ArchiveFacade делегує виконання операцій:

- Адаптерам (ArchiveAdapter) для створення, додавання, видалення та розпакування архівів.
- Фабрикам (ArchiveTesterFactory) для перевірки цілісності архівів.
- Стратегіям (FileProcessingStrategy) для обробки файлів.

Ця взаємодія дозволяє клієнту працювати з архівами різних форматів та типів без занурення у внутрішні деталі системи.

```
/Users/sofia/Library/Java/JavaVirtualMachines/openjdk-19.0.2/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA.app/Contents/l
Verifying file checksum:
Calculated checksum: e7cb632359a2be17c1008b50f9ec85691cd5d66834d5fe8f63ef65ceb06682ee
File checksum is valid for: example.txt

Deleting file:
File deleted: example.txt

Creating archive:
Archive created: archive.zip

Extracting archive:
Extracting .zip archive: archive.zip to ./output

Adding file to archive:
Adding file: example.txt to archive: archive.zip

Deleting file from archive:
Deleting file: example2.txt from archive: archive.zip

Testing archive integrity:
Testing .zip archive: archive.zip
Archive archive.zip is valid.

Demo completed!
```

Рис 2. Застосування шаблону при реалізації програми

Робота патерну

Перевірка контрольної суми файлу

Роль фасаду: фасад викликає стратегію `ChecksumVerificationStrategy`, яка обчислює контрольну суму файлу example.txt та перевіряє її валідність.

Спрощення: клієнтський код не знає про деталі обчислення checksum (використання ChecksumUtils), оскільки це приховано фасадом.

Видалення файлу

Роль фасаду: фасад змінює стратегію на `FileDeletionStrategy` та делегує видалення файлу стратегії.

Спрощення: фасад дозволяє легко переключатися між стратегіями, забезпечуючи однаковий інтерфейс для різних операцій обробки файлів.

Створення архіву

Роль фасаду: фасад викликає метод `createArchive`, який через адаптер `ZipArchiveAdapter` створює ZIP-архів із зазначеними файлами.

Спрощення: використання адаптера через інтерфейс `ArchiveAdapter` приховує специфічну логіку роботи з архівами.

Розпакування архіву

Роль фасаду: фасад делегує розпакування архіву через метод `extractArchive` адаптера ZipArchiveAdapter.

Спрощення: клієнтський код не знає, як реалізована розпаковка – це залишається завданням адаптера.

Додавання файлу до архіву

Роль фасаду: фасад використовує метод `addFileToArchive`, який викликає функціональність адаптера для додавання файлу до архіву.

Спрощення: адаптер реалізує операцію додавання файлів у форматі ZIP, але фасад уніфікує доступ до цієї функціональності.

Видалення файлу з архіву

Роль фасаду: метод `deleteFileFromArchive` фасаду викликає логіку видалення файлу через адаптер ZipArchiveAdapter.

Спрощення: клієнтський код залишається простим і чистим, оскільки деталі видалення приховані.

Тестування архіву на цілісність

Роль фасаду: фасад використовує фабрику `ZipArchiveTesterFactory`, щоб створити тестувальник ZipArchiveTester, який перевіряє архів на цілісність.

Спрощення: фасад забезпечує абстракцію створення тестувальника, не прив'язуючи клієнтський код до конкретного типу архіву.

Клас `ArchiveFacade` надає уніфікований доступ до складних операцій над архівами та файлами. Фасад приховує деталі роботи адаптерів, стратегій і фабрик, спрощуючи клієнтський код. Клієнтський код працює лише з фасадом, не взаємодіючи напряму з класами підсистеми. Фасад є “посередником” між клієнтом і підсистемами, забезпечуючи простоту використання та зручність розширення.

Переваги використання шаблону Facade у цьому випадку

1. Спрощення клієнтського коду

Шаблон "Facade" надає єдиний інтерфейс для виконання операцій над архівами та файлами. Усі складні дії (створення, розпакування, перевірка архівів, додавання та видалення файлів) виконуються через ArchiveFacade, що робить клієнтський код зрозумілим і чистим.

2. Інкапсуляція складності

Деталі реалізації операцій приховані за фасадом. Фасад делегує роботу відповідним адаптерам (Adapter), фабрикам (Factory Method) і стратегіям (Strategy), не розкриваючи клієнтові внутрішню структуру підсистеми. Це дозволяє клієнтському коду працювати лише з фасадом.

3. Гнучкість і розширюваність

- Додавання нових форматів архівів (.tar.gz, .rar, .zip, .ace) здійснюється через Adapter, не вимагаючи змін у класі ArchiveFacade.
- Додавання нових операцій обробки файлів (наприклад, шифрування, перевірка метаданих) реалізується за допомогою Strategy, що дозволяє розширити функціональність фасаду без його зміни.

4. Зменшення залежностей

Клієнтський код не залежить від конкретних реалізацій адаптерів чи стратегій. Замість цього він взаємодіє лише з фасадом. Це забезпечує низьке зв'язування системи та полегшує її супровід.

5. Покращення підтримки та масштабованості

Фасад спрощує додавання нових функцій або модифікацію існуючих компонентів. Наприклад:

- Додавання нового адаптера для архіву .7z не потребує змін у фасаді.
- Додавання нової стратегії для роботи з файлами (наприклад, зашифрувати файл) можна інтегрувати, не змінюючи існуючий код.

6. Єдиний контроль точок доступу

Фасад централізує доступ до функцій роботи з архівами, що забезпечує уніфіковану логіку виконання операцій і мінімізує можливі помилки в коді.

7. Зручність тестування

Оскільки всі операції виконуються через ArchiveFacade, тестування функціоналу архіватора стає простішим. Тестування фасаду автоматично охоплює взаємодію з адаптерами, фабриками та стратегіями.

8. Мінімізація дублікатів коду

Використання фасаду дозволяє уникнути повторення коду для однакових операцій, таких як створення або перевірка архівів. Усі виклики об'єднані в одному місці.

Висновок: отже, у ході виконання лабораторної роботи було реалізовано шаблон проектування Facade, що надав єдиний інтерфейс для роботи з архівами різних форматів (.zip, .rar, .tar.gz, .ace). Фасад спростив клієнтський код, приховавши складну логіку підсистеми, а також забезпечив гнучкість і можливість розширення за рахунок інтеграції шаблонів Adapter, Factory Method та Strategy. Завдяки цьому система стала зрозумілою, легко підтримуваною та масштабованою, а клієнтська взаємодія з функціоналом архівування стала зручною та структурованою.

Посилання на код: https://github.com/SofiiaKashubiak/Archivator-project.git