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 №5

Технології розроблення програмного забезпечення

ШАБЛОНИ «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»

Варіант №14

Виконав:

студент групи ІА-14,

Міщук Максим Дмитрович

Перевірив:

Мягкий М.Ю.

Київ 2023

Тема роботи: Шаблони «Adapter», «Builder», «Command», «Chains of responsibility», «Prototype».

Вхідні дані:

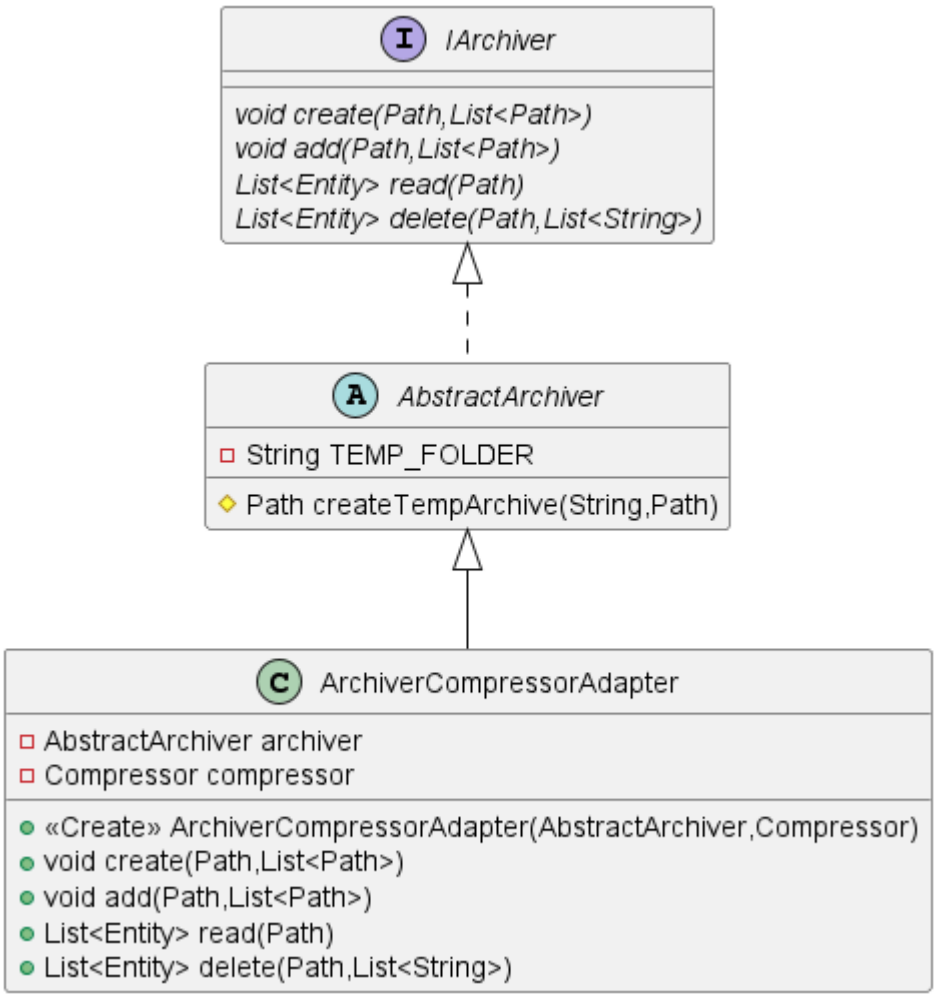
..14 Архіватор (strategy, adapter, factory method, facade, visitor, p2p)

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.

Хід роботи:

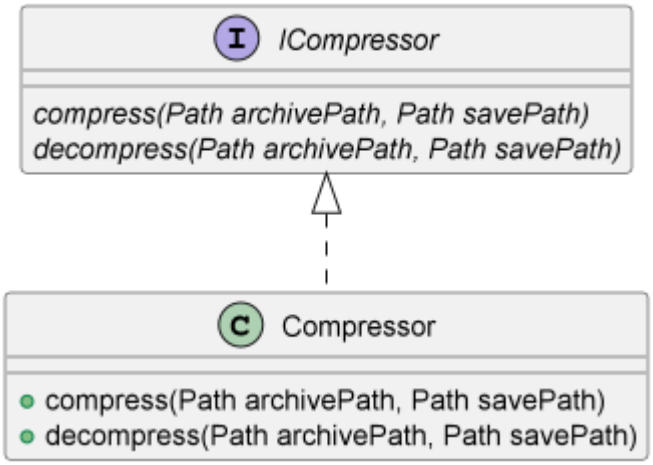
Pattern Adapter – це структурний патерн проектування, що дає змогу об’єктам із несумісними інтерфейсами працювати разом (взято з Refactoring.guru). Суть полягає в створенні, обгортки, що спадкується від одного з двох несумісних компонентів, а приймає в себе протилежний компонент, тим самим створюється так званий «перехідник».

Так наприклад архіви типу TAR можуть бути додатково стиснуті різними компресорами, тоді перед тим як прийняти та відправити такий архів, його треба стиснути та розтиснути відповідно. За для тоги аби поведінки роботи з такими архівами не відрізнялася від інших, було створено адаптер що спадкується від звичайного архіву й приймає в себе компресор та базовий архів та в середині робить з ними додаткові дії. Далі наведено діаграму класів цієї частини:



Діаграми класів адаптеру

Клас спадкується від абстрактного архіватора та при ініціалізації приймає в себе інший архіватор та компресор. Методи реалізовані поєднанням дій описаних раніше класів: результат роботи компресора передається до архіватора й у зворотному напрямку. Детально про *IArchiver*, *AbstractArchiver* та їх реалізації буде описано в 7-й лабораторній роботі. А зараз буде показано діаграму класів компресора та його адаптеру:



Діаграма класів додаткового стиснення

Код реалізації:

```
public interface ICompressor {
    void compress(Path archivePath, Path savePath) throws IOException, CompressorException;
    void decompress(Path archivePath, Path savePath) throws IOException, CompressorException;
}
```

ICompressor.java

```java
public class Compressor implements ICompressor {
    private final CompressorType type;

    public Compressor(CompressorType type) {
        this.type = type;
    }

    @Override
    public void compress(Path archivePath, Path savePath) throws IOException, CompressorException
{
        try (
                BufferedInputStream bis = new
BufferedInputStream(Files.newInputStream(archivePath));
                BufferedOutputStream bos = new
BufferedOutputStream(Files.newOutputStream(savePath));
                CompressorOutputStream cos = new
CompressorStreamFactory().createCompressorOutputStream(type.name(), bos)
        ) {
            IOUtils.copy(bis, cos);
        }
    }

    @Override
    public void decompress(Path archivePath, Path savePath) throws IOException,
CompressorException {
        try (
                BufferedOutputStream bos = new
BufferedOutputStream(Files.newOutputStream(savePath));
                BufferedInputStream bis = new
BufferedInputStream(Files.newInputStream(archivePath));
                CompressorInputStream cis = new
CompressorStreamFactory().createCompressorInputStream(type.name(), bis)
        ) {
            IOUtils.copy(cis, bos);
        }
    }
}
```

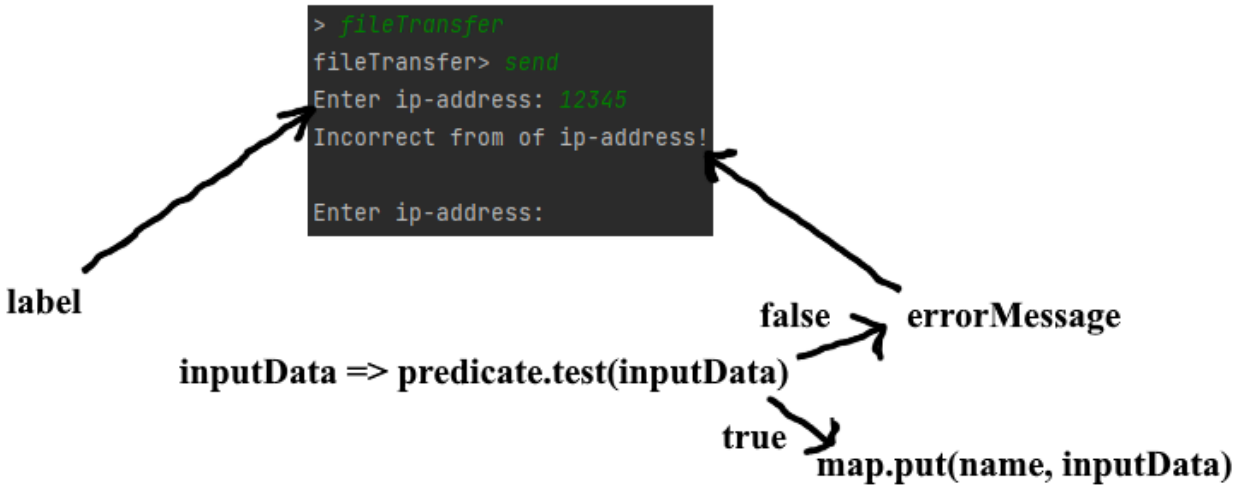
Compressor.java

```java
public class ArchiverCompressorAdapter extends AbstractArchiver {
    private final AbstractArchiver archiver;
    private final Compressor compressor;
    public ArchiverCompressorAdapter(AbstractArchiver archiver, Compressor compressor) {
        this.archiver = archiver;
        this.compressor = compressor;
    }

    @Override
    public void create(Path archivePath, List<Path> filePaths) throws IOException, ArchiveException {
        Path tempPath = createTempArchive("compress", archivePath);
        archiver.create(tempPath, filePaths);
        try {
            compressor.compress(tempPath, archivePath);
        } catch (CompressorException e) {
            throw new RuntimeException(e);
        } finally {
            Files.delete(tempPath);
        }
    }

    @Override
    public void add(Path archivePath, List<Path> filePaths) throws IOException, ArchiveException {
        Path tempPath = createTempArchive("compress", archivePath);
        try {
            compressor.decompress(archivePath, tempPath);
            archiver.add(tempPath, filePaths);
            compressor.compress(tempPath, archivePath);
        } catch (CompressorException e) {
            throw new RuntimeException(e);
        } finally {
            Files.delete(tempPath);
        }
    }

    @Override
    public List<Entity> read(Path archivePath, ArchiveReadingType archiveReadingType) throws IOException,
ArchiveException {
        Path tempPath = createTempArchive("compress", archivePath);
        try {
            compressor.decompress(archivePath, tempPath);
            return archiver.read(tempPath, archiveReadingType);
        } catch (CompressorException e) {
            throw new RuntimeException(e);
        } finally {
            Files.delete(tempPath);
        }
    }

    @Override
    public List<Entity> delete(Path archivePath, List<String> fileNames) throws IOException,
ArchiveException {
        Path tempPath = createTempArchive("compress", archivePath);
        List<Entity> deletedEntities;
        try {
            compressor.decompress(archivePath, tempPath);
            deletedEntities = archiver.delete(tempPath, fileNames);
            compressor.compress(tempPath, archivePath);

            return deletedEntities;
        } catch (CompressorException e) {
            throw new RuntimeException(e);
        } finally {
            Files.delete(tempPath);
        }
    }
}
```

ArchiverCompressorAdapter.java

Додатково:

Для полегшення створення та видозміни форм на стороні користувацького інтерфейсу було використано шаблон будівельник. Найменшою одиницею в даній системі є поле, куди користувач записує дані. Для кращого розуміння нижче знаходиться приклад, що відповідаю візуальному відображенню поля:



Приклад поля

Кожне поле складається з етикетки (*label*), де вказується інформація перед очікуванням введення від користувача. Предикатора (*predicate*), що перевіряє введені дані на задані умови під час ініціалізації. Повідомлення про помилку (*errorMessage*), якщо умови предикатора не було виконано. Також кожне поле містить власне ім’я (*name*) для витягнення даних цих полів під час виконання повноцінних форм.

Форми складаються зі списку полів та словника де парою ключ-значення виступає рядок-рядок. Під час ініціалізації ззовні приходить список полів та створюється пустий словник результатів. Для того щоб запустити форму викликається метод *execute()*, що повертає у відповідь мапу (словник) значень.

Однак, можна побачити, що створення форм в ручному вигляді є не дуже зручним. Це відбувається через монотонне створення полів , кожне з яких має

в собі велику кількість параметрів при створенні. В додаток, такі списки буде дуже важно видозмінювати, в разі потреби.

Тому для полегшення розробки було створено клас *FormBuilder*. Якщо спростити, то білдер складається з таких функцій: встановити потік вводу та принтер (за замовчуванням вони приймають в себе значення стандартних потоків вводу/виводу системи) і додаванням полів. Останнє присутнє в трьох варіантах: напряму додати поле, створити й додати поле що приймає в себе одне значення, створити й додати поле що приймає в себе декілька значень. Після того як список був остаточно сформований, треба здійснити виклик методу *build()*, що поверне готову для використання форму.

Висновок:

За допомоги шаблону адаптер вдалося привести несумісні інтерфейси під один стандарт. Таким чином архіви з додатковим стисненням працюють від того ж самого інтерфейсу що і звичайні архіви, що робить загальну логіку застосунку більш природньою та зрозумілою.

Додатково було використано шаблон будівельник, в реалізації вхідних форм. Таким чином вигляд створення полів має більш читабельний вигляд, де кожен атрибут додається крок за кроком, а сам код в разі потреби легше видозмінювати.

Посилання на репозиторій: Maxim-Mishchuk/trpz-archiver at develop (github.com)