МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Технології розроблення програмного забезпечення:

Лабораторний практикум

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра за освітньою програмою «Інтегровані інформаційні системи» за спеціальністю
126 «Інформаційні системи та технології»

Київ
КПІ ім. Ігоря Сікорського
2025

Технології розроблення програмного забезпечення: Лабораторний практикум [Електронний ресурс] : навч. посіб. для здобувачів ступеня бакалавра за освітньою програмою «Інтегровані інформаційні системи» за спеціальністю 126 «Інформаційні системи та технології», / КПІ ім. Ігоря Сікорського; укладачі: О.А. Амонс., М.Ю. Мягкий – Електронні текстові дані (1 файл: 2 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2025. – 113 с.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського

(протокол № 5 від 06 березня 2025 р.) за поданням Вченої ради факультету Інформатики та обчислювальної техніки (протокол № 8 від 24 лютого.2025 р.)

Електронне мережне навчальне видання

Технології розроблення програмного забезпечення:

Лабораторний практикум

Укладачі:	Амонс Олександр Анатолійович, к.т.н.
	Мягкий Михайло Юрійович
Відповідальний редактор:	Букасов М.М., к.т.н., доцент кафедри інформаційних систем та технологій КПІ ім. Ігоря Сікорського
Рецензент:	Крамар Ю.М. к.т.н., доц., доцент кафедри інформатики та програмної інженерії КПІ ім. Ігоря Сікорського

Навчальний посібник охоплює теоретичний матеріал та практичні завдання, які необхідні для виконання лабораторного практикуму з дисципліни «Технології розроблення програмного забезпечення». Практикум охоплює повний цикл проєктування програмних систем та практичне використання патернів проєктування при реалізації спроєктованої системи. В посібнику наведені теоретичні відомості для практичного оволодіння навичками проєктування програмної системи з використанням мови UML, використання патернів проєктування для вирішення практичних задач, реалізації спроєктованої системи, надані теоретичні відомості для практичного застосування в реалізації систем технологій розподіленої взаємодії, таких як peer-to-peer, SOA та мікросервіси.

Зміст

ВСТУП

Даний лабораторний практикум повністю охоплює силабус (робоча програма навчальної дисципліни) дисципліни «Технології розробки програмних систем» та формує у студентів відповідні знання, навички та вміння, відповідно до вимог освітньої програми підготовки фахівця за спеціальністю 126 – Інформаційні системи та технології.

Лабораторні роботи побудовані з використанням основ проєктування систем мовою UML, розуміння та використання в проєктуванні та реалізації систем патернів проєктування, що включає в себе збір вимог та проєктування діаграм варіантів використання, сценаріїв використання та проєктування поведінки системи у вигляді діаграм послідовностей та станів, проєктування статичної структури системи починаючи від діаграм розгортання до специфікації архітектури системи та її частин у вигляді діаграм класів на різних рівнях, а також реалізацію спроєктованої системи на вибраній мові програмування.

ЗАГАЛЬНІ ПОЛОЖЕННЯ

Вимоги до виконання лабораторних робіт

Відповідно до обраної теми кожен студент повинен спроєктувати діаграми UML відповідно до лабораторної роботи та реалізувати у вигляді програмного коду фрагмент програми, що демонструє навички та знання відповідно до по-поставленого завданням роботи. Програмний код повинен відповідати вимогам функціональності відповідно до теми лабораторного курсу.

Вимоги до оформлення звіту

Звіт в електронному вигляді має містити титульний аркуш (з номером і темою лабораторної роботи, прізвища та ініціалів студента, який її виконав), назва та номер теми обраної для лабораторного циклу, короткі теоретичні відомості, діаграми UML для другої та третьої лабораторних робіт або діаграми класів на мові UML реалізованого шаблону програмування в термінах власної теми розробки для всіх інших, релевантні фрагменти програмного коду, висновки по лабораторній роботі.

1. ЛАБОРАТОРНА РОБОТА № 1

Тема: Системи контроля версій. Розподілена система контролю версій «Git».
Мета: Навчитися виконувати основні операції в роботі з децентралізованими системами контролю версій на прикладі роботи з сучасною системою Git.

1.1. Завдання

- Ознайомитись із короткими теоретичними відомостями.
- Створити Git репозиторій.
- Клонувати Git репозиторій.
- Продемонструвати базову роботу з репозиторієм: створення версій, додавання тегів, робіт з гілками (створення та злиття), робота з комітами, вирішення конфліктів, а також робота з віддаленим репозиторієм.

1.2. Теоретичні відомості

1.2.1. Призначення систем управління версіями

Система управління версіями (від англ. Version Control System або Source Control System) – програмне забезпечення яке призначено допомогти команді розробників керувати змінами в вихідному коді під час роботи [1]. Система керування версіями дозволяє додавати зміни в файлах в репозиторій і таким чином після кожної фіксації змін мани нову ревізію файлів. Це дозволяє повертатися до попередніх версій коду для аналізу внесених змін або пошуку, які зміни привели до появи помилки. Таким чином можна знайти хто, коли і які зміни зробив в коді, а також чому ці зміни були зроблені.

Такі системи найбільш широко використовуються при розробці програмного забезпечення для зберігання вихідних кодів програми, що розробляється. Однак вони можуть з успіхом застосовуватися і в інших областях, в яких ведеться робота з великою кількістю електронних документів, що безперервно змінюються. Зокрема, системи керування версіями застосовуються

у САПР, зазвичай у складі систем керування даними про виріб (PDM). Керування версіями використовується у інструментах конфігураційного керування (Software Configuration Management Tools).

1.2.2. Історія розвитку систем контролю версій

Умовно, розвиток систем контролю версій можна розбити на наступні етапи: ранній етап, етап централізованих систем, етап децентралізації та етап хмарних платформ.

Ранній етап

На цьому етапі основна увага приділялася роботі з окремими файлами у локальному середовищі.

Найпершою системою контролю версій була система «скопіювати і вставити», коли більшість проєктів просто копіювалася з місця на місце зі зміною назва (проєкт_1; проєкт_новий; проєкт_найновіший і т.д.), як правило у вигляді zip архіву або подібних (arj, tar). Звичайно, такі маніпуляції над файловою системою навряд чи можна назвати хоч скільки повноцінною системою контролю версій (або системою взагалі). Для вирішення цих проблем 1982 року з'являється RCS.

RCS

Однією з основних нововведень RCS було використання дельт для зберігання змін (тобто зберігаються ті рядки, які змінилися, а не весь файл). Однак він мав низку недоліків.

Насамперед він був тільки для текстових файлів. Не було центрального репозиторію; кожен версіонований файл мав власний репозиторій як rcs файлу поруч із самим файлом. Тобто якщо на проєкті було 100 файлів, поруч лягало 100 rcs файлів. У кращому випадку ці 100 файлів утворювалися в директорії RCS (при правильному налаштуванні). Найменування версій і гілок було неможливим.

Етап централізованих систем

На початку 90-х почалася епоха централізованих систем контролю версій. У цей період розробники почали переходити до централізованих систем, що дозволяли працювати кільком користувачам одночасно через сервер

Одина із перших найпопулярніших систем (і досі використовувана) система контролю версій – CVS. Цю епоху можна охарактеризувати досить сформованим уявленням про системи контролю версій, їх можливості, появою центральних репозиторіїв (та синхронізації дій команди).

SVN

SVN – у порівнянні з CVS це був наступний крок. Надійна та швидкодіюча систему контролю версій, яка зараз розробляється в рамках проєкту Apache Software Foundation. Вона реалізована за технологією клієнт-сервер та відрізняється неймовірною простотою – дві кнопки (commit, update). Порівняно з CVS, це удосконалена централізована система з кращим управлінням комітами та резервними копіями.

Незважаючи на це, SVN дуже погано вміє створювати та зливати гілки та погано вирішує конфліктні ситуації з версіями. Але, в багатьох проєктах до цих пір використовується SVN.

Етап децентралізації

Децентралізовані системи усунули залежність від центрального сервера та дозволили кожному розробнику мати повну копію репозиторію.

У 1992 році з'явився один з основних представників світу систем розподіленого контролю версій. ClearCase був однозначно попереду свого часу і для багатьох він досі є однією з найпотужніших систем контролю версій будь-коли створених.

Дана система дозволяла користуватися віртуальною файловою системою для зберігання та отримання змін; мала широкий діапазон повноважень щодо зміни, впровадження у процес розробки (аудит збірок товару, версії, зливання змін, динамічні уявлення); запускалася на безлічі різних систем.

У 2005 році було створено дві знакові системи контролю версій Git та Mercurial. Вони стали революційними системами, які забезпечили швидкість, надійність і гнучкість роботи. Вони мають багато ідентичних команд, хоча «під капотом» вони мають різні підходи до реалізації. Досить довго вони конкурували одна з одною, але починаючи з 2018 Git поступово виходить на лідерську позицію серед безкоштовних систем контролю версій.

Git

Лінус Торвальдс, т.зв. Батько Лінукса, розробив і впровадив першу версію Гіт для надання можливості розробникам ядра Лінукс проводити контроль версій не тільки в BitKeeper.

Гіт є системою розподіленого контролю версій, коли кожен розробник має власний репозиторій, куди він вносить зміни [2]. Далі система гіт синхронізує репозиторії із центральним репозиторієм. Це дозволяє проводити роботу незалежно від центрального репозиторію (на відміну від SVN, коли версіонування передбачало наявність зв'язку з центральним сервером), перекладає складності ведення гілок та склеювання змін більше на плечі системи, ніж розробників та ін.

Зміни зберігаються у вигляді наборів змін (changeset), що отримує унікальний ідентифікатор (хеш-сума на основі самих змін).

Mercurial

Mercurial був створений як і Git після оголошення про те, що BitKeeper більше не буде безкоштовним для всіх. Багато в чому схожий на Git, Mercurial також використовує ідею наборів змін, але на відміну від Git, зберігає їх у не у вигляді вузла в графі, а вигляді плоского набору файлів і папок, званих revlog.

Етап хмарних платформ

Приблизно з 2010 року і до цих пір також можна виділити етап хмарних платформ, основним лозунгом яких є «Інтеграція та автоматизація».

У сучасну епоху акцент робиться на інтеграції систем контролю версій із хмарними платформами та автоматизації розробки. І в більшості випадків такою системою контролю версій є Git.

Можна виділити такі ключові хмарні платформи на основі Git: GitHub, GitLab, Bitbucket. Вони підтримують CI/CD, спільну роботу та інтеграції, інструменти для DevOps, аналітики та автоматичного тестування.

Таким чином, основною характеристикою цього етапу є інтеграція систем контролю версій в хмарні сервіси для глобальної співпраці, які додатково підтримують розширену функціональність для автоматизації процесів та інтеграції з іншими сервісами.

1.2.3. Робота з Git

Робота з Git може виконуватися з командного рядка, а також за допомогою візуальних оболонок. Командний рядок використовується програмістами, як можливість виконання всіх доступних команд, а також можливості складання складних макросів. Візуальні оболонки як правило дають більш наглядне представлення репозиторію у вигляді дерева, та більш зручний спосіб роботи з репозиторієм, але, дуже часто доступний не весь набір команд Git, а лише саме ті, що найчастіше використовуються.

Прикладами візуальних оболонок для роботи з Git є Git Extension, SourceTree, GitKraken, GitHub Desktop та інші.

Основна ідея Git, як і будь-якої іншої розподіленої системи контроля версій – кожен розробник має власний репозиторій, куди складаються зміни (версії) файлів, та синхронізація між розробниками виконується за допомогою синхронізації репозиторіїв. Процес роботи виглядає так, як зображено на рисунку 1.1.

Відповідно, є ряд основних команд для роботи [2]:

1. Клонувати репозиторій (git clone) – отримати копію репозиторію на локальну машину для подальшої роботи з ним;
2. Синхронізація репозиторіїв (git fetch або git pull) – отримання змін із віддаленого (вихідного, центрального, або будь-якого іншого такого ж) репозиторію;

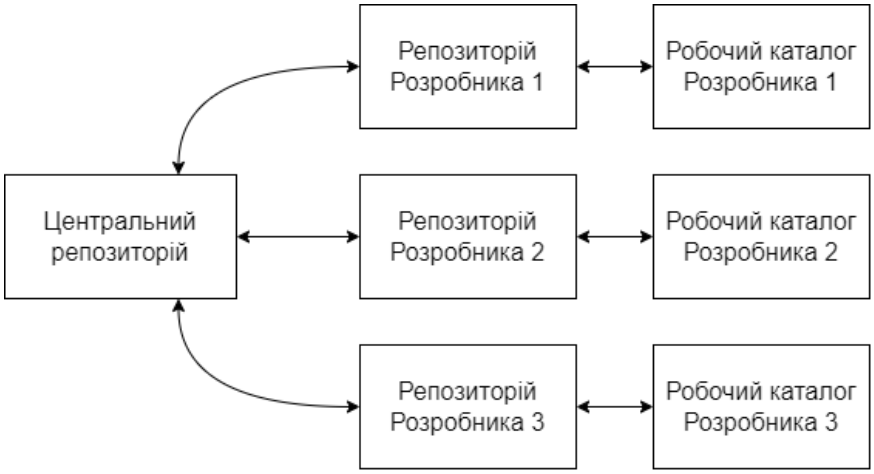

Рисунок 1.1. Схема процесу роботи з Git

3. Фіксація змін в репозиторій (git commit) – фіксація виконаних змін в програмному коді в локальний репозиторій розробника;

4. Синхронізація репозиторіїв (git push) – переслати зміни – push – передача власних змін до віддаленого репозиторію – Записати зміни – commit – створення нової версії;

5. Оновитись до версії – update – оновитись до певної версії, що є у репозиторії.

6. Об’єднання гілок (git merge) – об’єднання вказаною гілки в поточну (часто ще називається «злиттям»).

Таким чином, якщо розглядати основний робочий процес програміста в команді, то він виглядає наступним чином: На початку роботи з проєктом виконується клонування, після цього, в рамках виконання поставленої задачі, створюється бранч і всі зміни в коді, зроблені в рамках цієї задачі фіксуються в репозиторії (періодично виконується синхронізація з основним репозиторієм). Далі, коли задача виконана, то виконується об’єднання гілки з основною гілкою і фінальна синхронізація з центральним репозиторієм.

1.2.4 Додаткові матеріали для самостійного опрацювання

1. https://learngitbranching.js.org/ – інтерактивний посібник по командах гіт. Можете пройти кілька перших уроків, чого може бути достатньо для опанування базових команд, але по бажанню можна і все пройти.

2. https://www.codecademy.com/learn/learn-git – ще один інтерактивний курс

3. https://git-scm.com/doc – документація

4. Допомога від самого Git, наприклад, команда в консолі:

```
git help branch
```

або:

```
git branch -h
```

1.3. Приклад виконання завдання

1. Створити локальний репозиторій.

cmd

```
PS D:\Projects\Sandbox> git init lab1
Initialized empty Git repository in
D:/Projects/Sandbox/lab1/.git/
```

2. Додати довільний файл з довільним текстом (в даному випадку текст – test).

cmd

```
PS D:\Projects\Sandbox> cd lab1
PS D:\Projects\Sandbox\lab1> echo "test" > hello.txt
```

3. Зафіксувати додавання файлу.

cmd

```
PS D:\Projects\Sandbox\lab1> git add hello.txt
PS D:\Projects\Sandbox\lab1> git commit -m "Hello added"
[master (root-commit) 52dcdc8] Hello added
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 hello.txt
```

4. Додати нову директорію з довільним ім'ям.

cmd

```
PS D:\Projects\Sandbox\lab1> mkdir child_dir


    Directory: D:\Projects\Sandbox\lab1


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          9/7/2024   1:13 PM
child_dir
```

5. Додати файл у директорію.

cmd

```
PS D:\Projects\Sandbox\lab1> cd child_dir
PS D:\Projects\Sandbox\lab1\child_dir> echo "inner" >
inner.txt
```

6. Зафіксувати додавання директорії із файлом.

cmd

```
PS D:\Projects\Sandbox\lab1\child_dir> cd ..
PS D:\Projects\Sandbox\lab1> git add child_dir
PS D:\Projects\Sandbox\lab1> git commit -m "Add child
directory with content"
[master a37cf9f] Add child directory with content
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 child_dir/inner.txt
```

7. Створити гілку і перейти на неї.

cmd

```
PS D:\Projects\Sandbox\lab1> git branch feature_branch
PS D:\Projects\Sandbox\lab1> git checkout feature_branch
Switched to branch 'feature_branch'
```

8. Видалити додану директорію і зафіксувати зміни.

cmd

```
PS D:\Projects\Sandbox\lab1> rmdir child_dir -r
PS D:\Projects\Sandbox\lab1> git add .
PS D:\Projects\Sandbox\lab1> git commit -m "Remove
child_dir"
[feature_branch 839d32f] Remove child_dir
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 child_dir/inner.txt
```

9. Злити зміни з основною гілкою.

cmd

```
PS D:\Projects\Sandbox\lab1> git checkout master
Switched to branch 'master'
PS D:\Projects\Sandbox\lab1> git merge feature_branch
Updating a37cf9f..839d32f
Fast-forward
 child_dir/inner.txt | Bin 16 -> 0 bytes
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 child_dir/inner.txt
```

10. Вивести історію на екран.

cmd

```
PS D:\Projects\Sandbox\lab1> git log
commit 839d32f5fc0f30a5be9254ca35c8667e6b4e196b (HEAD ->
master, feature_branch)
Author: Test User <test.user@gmail.com>
Date:   Thu Sep 7 13:15:24 2024 +0300

    Remove child_dir

commit a37cf9fb865342eadb3e834bf7a3bff191ae9779
Author: Test User <test.user@gmail.com>
Date:   Thu Sep 7 13:14:17 2024 +0300

    Add child directory with content

commit 52dcdc8e18fdc1284982003e9815b036e98ad686
Author: Test User <test.user@gmail.com>
```

```
Date:   Thu Sep 7 13:13:19 2024 +0300

    Hello added
```

Отже, кінцевим результатом лабораторної роботи повинні стати:

1. Створений локальний репозиторій.
2. Створений і закомічений в репозиторій файл.
3. Створена в проєкті директорія.
4. В репозиторії існує дві гілки.
5. Гілка зі змінами злита з основною гілкою

1.4. Питання до лабораторної роботи

1. Що таке система контролю версій (СКВ)?
2. Поясніть відмінності між розподіленою та централізованою СКВ.
3. Поясніть різницю між stage та commit в Git.
4. Як створити гілку в Git?
5. Як створити або скопіювати репозиторій Git з віддаленого серверу?
6. Що таке конфлікт злиття, як створити конфлікт, як вирішити конфлікт?
7. В яких ситуаціях використовуються команди: merge, rebase, cherry-pick?
8. Як переглянути історію змін Git репозиторію в консолі?
9. Як створити гілку в Git не використовуючи команду git branch?
10. Як підготувати всі зміни в поточній папці до коміту?
11. Як підготувати всі зміни в дочірній папці до коміту?
12. Як переглянути перелік наявних гілок в репозиторії?
13. Як видалити гілку?
14. Які є способи створення гілки та в чому між ними різниця?

2. ЛАБОРАТОРНА РОБОТА № 2

Тема: Основи проектування.

Мета: Обрати зручну систему побудови UML-діаграм та навчитися будувати діаграми варіантів використання для системи що проєктується, розробляти сценарії варіантів використання та будувати діаграми класів предметної області.

2.1. Завдання

• Ознайомитись з короткими теоретичними відомостями.

• Проаналізувати тему та спроєктувати діаграму варіантів використання відповідно до обраної теми лабораторного циклу.

• Спроєктувати діаграму класів предметної області.

• Вибрати 3 варіанти використання та написати за ними сценарії використання.

• На основі спроєктованої діаграми класів предметної області розробити основні класи та структуру бази даних системи. Класи даних повинні реалізувати шаблон Repository для взаємодії з базою даних.

• Нарисувати діаграму класів для реалізованої частини системи.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму варіантів використання відповідно, діаграму класів системи, вихідні коди класів системи, а також зображення структури бази даних.

2.2. Теоретичні відомості

2.2.1 Вступ до мови UML

Мова UML є загальноцільовою мовою візуального моделювання, яка розроблена для специфікації, візуалізації, проєктування та документування компонентів програмного забезпечення, бізнес-процесів та інших систем [3]. Мова UML є досить строгим та потужним засобом моделювання, який може бути ефективно використаний для побудови концептуальних, логічних та графічних

моделей складних систем різного цільового призначення. Ця мова увібрала в себе найкращі якості та досвід методів програмної інженерії, які з успіхом використовувалися протягом останніх років при моделюванні великих та складних систем.

З погляду методології ООАП (об'єктно-орієнтованого аналізу та проєктування) досить повна модель складної системи є певною кількістю взаємопов'язаних уявлень (views), кожне з яких відображає аспект поведінки або структури системи. У цьому найзагальнішими уявленнями складної системи прийнято вважати статичне і динамічне, які у своє чергу можуть поділятися інші більш приватні.

Принцип ієрархічної побудови моделей складних систем передбачає розгляд процес побудови моделей на різних рівнях абстрагування або деталізації в рамках фіксованих уявлень.

Рівень представлення (layer) – спосіб організації та розгляду моделі на одному рівні абстракції, що представляє горизонтальний зріз архітектури моделі, тоді як розбиття представляє її вертикальний зріз.

При цьому вихідна або початкова модель складної системи має найбільш загальне уявлення та відноситься до концептуального рівня. Така модель, що отримала назву концептуальної, будується на початковому етапі проєктування і може не містити багатьох деталей та аспектів системи, що моделюється. Наступні моделі конкретизують концептуальну модель, доповнюючи її уявленнями логічного та фізичного рівня.

Загалом процес ООАП можна розглядати як послідовний перехід від розробки найбільш загальних моделей та уявлень концептуального рівня до більш приватних і детальних уявлень логічного та фізичного рівня. У цьому кожному етапі ООАП дані моделі послідовно доповнюються дедалі більше деталей, що дозволяє їм адекватно відбивати різні аспекти конкретної реалізації складної системи.

В рамках мови UML уявлення про модель складної системи фіксуються у вигляді спеціальних графічних конструкцій, що отримали назву діаграм.

Діаграма (diagram) – графічне уявлення сукупності елементів моделі у формі зв'язкового графа, вершинам і ребрам (дугам) якого приписується певна семантика. Нотація канонічних діаграм є основним засобом розробки моделей мовою UML.

У нотації мови UML визначено такі види діаграм:

- варіантів використання (use case diagram);
- класів (class diagram);
- кооперації (collaboration diagram);
- послідовності (sequence diagram);
- станів (statechart diagram);
- діяльності (activity diagram);
- компонентів (component diagram);
- розгортання (deployment diagram).

Перелічені діаграми є невід'ємною частиною графічної нотації мови UML. Понад те, процес ООАП нерозривно пов'язаний з процесом побудови цих діаграм. При цьому сукупність побудованих таким чином діаграм є самодостатньою в тому сенсі, що в них міститься вся інформація, яка потрібна для реалізації проєкту складної системи.

Кожна з цих діаграм деталізує та конкретизує різні уявлення про модель складної системи у термінах мови UML. При цьому діаграма варіантів використання являє собою найбільш загальну концептуальну модель складної системи, яка є вихідною для побудови інших діаграм. Діаграма класів, за своєю суттю, логічна модель, що відбиває статичні аспекти структурної побудови складної системи.

Діаграми кооперації та послідовностей є різновидами логічної моделі, які відображають динамічні аспекти функціонування складної системи. Діаграми станів та діяльності призначені для моделювання поведінки системи. І, нарешті,

діаграми компонентів і розгортання служать уявлення фізичних компонентів складної системи і тому представляють її фізичну модель.

2.2.2. Діаграма варіантів використання (Use-Cases Diagram)

Діаграма варіантів використання (Use-Cases Diagram) – це UML діаграма за допомогою якої у графічному вигляді можна зобразити вимоги до системи, що розробляється [3]. Діаграма варіантів використання – це вихідна концептуальна модель проєктованої системи, вона не описує внутрішню побудову системи.

Діаграми варіантів використання призначені для:

- визначення загальної межі функціональності проєктованої системи;
- формулювання загальних вимоги до функціональної поведінки проєктованої системи;
- подальшої розробка вихідної концептуальної моделі системи (діаграми класів);
- створення основи для виконання аналізу, проєктування, розробки та тестування.

Діаграми варіантів використання є відправною точкою при збиранні вимог до програмного продукту та його реалізації. Дана модель будується на аналітичному етапі побудови програмного продукту (збір та аналіз вимог) і дозволяє бізнес-аналітикам отримати більш повне уявлення про необхідне програмне забезпечення та документувати його.

Діаграма варіантів використання складається з низки елементів. Основними елементами є: варіанти використання або прецеденти (use case), актор або дійова особа (actor) та відносини між акторами та варіантами використання (relationship).

2.2.3. Актори (actor)

Актором називається будь-який об'єкт, суб'єкт чи система, що взаємодіє з модельованою бізнес-системою ззовні для досягнення своїх цілей або вирішення

певних завдань [3]. Це може бути людина, технічний пристрій, програма або будь-яка інша система, яка служить джерелом впливу на систему, що моделюється. Актора можна розглядати як роль, яка виконується людиною в системі. Зображення акторів на діаграмах представлено на рисунку 2.1.

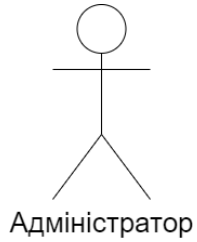

Рисунок 2.1. Зображення акторів на діаграмах UML

Ім'я актора має бути достатньо інформативним з точки зору програмного забезпечення, що розробляється, або предметної області. Для цього підходять найменування посад у компанії (наприклад, продавець, касир, менеджер, президент).

2.2.4. Варіанти використання (use case)

Варіант використання служить для опису служб, які система надає актору. Інакше кажучи кожен варіант використання визначає набір дій, здійснюваний системою під час діалогу з актором. Кожен варіант використання являє собою послідовність дій, який повинен бути виконаний системою, що проєктується при взаємодії її з відповідним актором, самі ці дії не відображаються на діаграмі.

Варіант використання відображається еліпсом, всередині якого міститься його коротке ім'я з великої літери у формі іменника або дієслова. Приклад зображення варіанта використання на діаграмі представлено на рисунку 2.2.

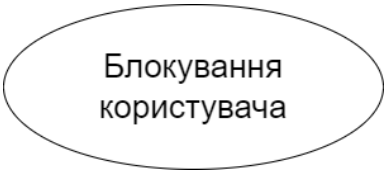

Рисунок 2.2. Позначення варіанта використання на діаграмах

При проєктуванні варіантів використання, потрібно приймати до уваги, що з назви варіанта використання має бути зрозуміла тривалість та результат його виконання.

Приклади варіантів використання: реєстрація, авторизація, оформлення замовлення, переглянути замовлення, перевірка стану поточного рахунку тощо.

2.2.5 Відносини на діаграмі варіантів використання

Відношення (relationship) – семантичний зв'язок між окремими елементами моделі.

Один актор може взаємодіяти з кількома варіантами використання. У цьому випадку цей актор звертається до кількох служб цієї системи. У свою чергу, один варіант використання може ініціюватися декількома акторами, надаючи для них свій функціонал.

Існують такі відносини: асоціації, узагальнення, залежність (складається з включення та розширення).

Асоціація (association) – узагальнене, невідоме ставлення між актором та варіантом використання.

Позначається суцільною лінією між актором та варіантом використання. Розрізняють ненаправлену (двонаправлену) асоціацію та однонаправлену асоціацію. Ненаправлена асоціація показує взаємодію без акцента на напрямок, або коли напрямок ще не аналізувався. Відображення ненаправленої асоціації між актором «Покупець» та варіантом використання «Перегляд списку доступних товарів» представлено на рисунку 2.3.

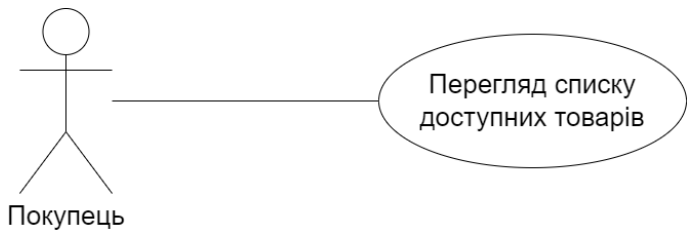

Рисунок 2.3. Ненаправлена асоціація

Спрямована, або направлена асоціація (directed association) – також показує що актор асоціюється з варіантом використання але показує, що варіант використання ініціалізується актором. Позначається стрілкою. Приклад направлених асоціацій представлено на рисунку 2.4.

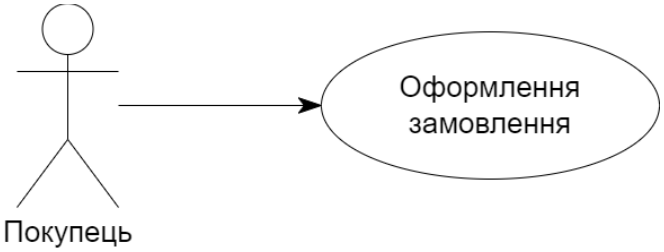

Рисунок 2.4. Приклад спрямованої асоціації

Спрямована асоціація дозволяє запровадити поняття основного актора (він є ініціатором асоціації) та другорядного актора (варіант використання є ініціатором, тобто передає акторові довідкові відомості або звіт про виконану роботу). Приклад асоціації направленої до актора представлено на рисунку 2.5.

Рисунок 2.5. Приклад асоціації направленої до актора

Відношення узагальнення (generalization) – показує, що нащадок успадковує атрибути у свого прямого батьківського елементу. Тобто, один елемент моделі є спеціальним або окремим випадком іншого елемента моделі. Може застосовуватися як до акторів, так і до варіантів використання.

Графічно відношення узагальнення позначається суцільною лінією зі стрілкою у формі незафарбованого трикутника, яка вказує на батьківський варіант використання.

На рисунку 2.6 показано відношення узагальнення. Варіант використання «Оплата замовлення» називається предком (чи батьком), а варіант використання «Оплата замовлення банківською картою» – нащадком (чи дочірнім) стосовно «Оплата замовлення».

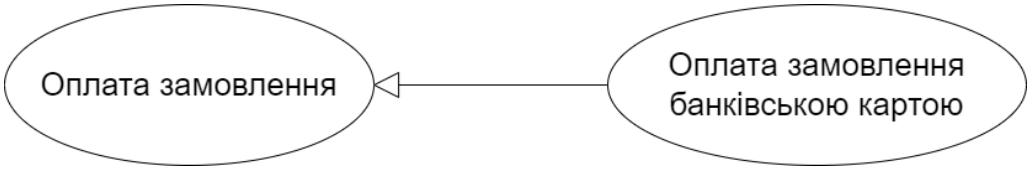

Рисунок 2.6. Відношення узагальнення

Особливості використання відношення узагальнення:

- головною особливістю відношення узагальнення є те, що воно може пов'язувати між собою лише елементи одного типу;
- один варіант використання може мати кілька батьківських варіантів використання (множинне успадкування);
- один варіант використання може бути предком кількох дочірніх варіантів використання (таксономічний характер відносин).

Також існує узагальнення між акторами, яке представлено на рисунку 2.7.

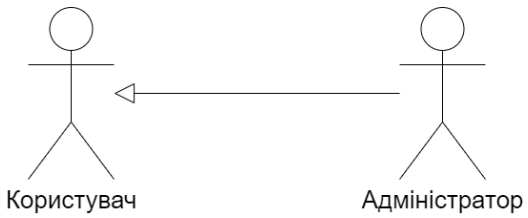

Рисунок 2.7. Відношення узагальнення між акторами

У наведеному прикладі Адміністратор успадковує всі атрибути свого предка Користувача, але може мати свої індивідуальні, які не зображені на рисунку.

Відношення залежності (dependency) визначається як форма взаємозв'язку між двома елементами моделі, призначена для специфікації тієї обставини, що зміна одного елемента моделі призводить до зміни деякого іншого елемента.

Загалом залежність є спрямованим бінарним ставленням, яке пов'язує між собою два елементи моделі: незалежний та залежний.

Відношення включення (include) – окремий випадок загального відношення залежності між двома варіантами використання, при якому деякий варіант використання містить поведінку, визначену в іншому варіанті використання.

Графічне зображення – пунктирна стрілка з стереотипом << include>>.

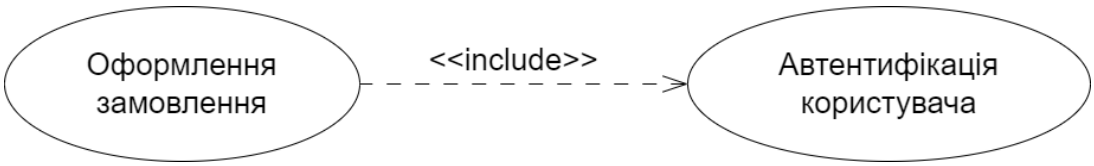

Рисунок 2.8. Відношення включення

Залежний варіант використання «Автентифікація користувача» називають базовим, а незалежний – включається («Оформлення замовлення»). На рисунку включення означає, що кожне виконання варіанта використання «Оформлення замовлення» завжди включатиме виконання варіанта використання «Автентифікація користувача». На практиці відношення включення використовується для моделювання ситуації, коли існує загальна частина поведінки двох або більше варіантів використання. Загальна частина виноситься на окремий варіант використання.

Особливості використання відношення включення:

- один базовий варіант використання може бути пов'язаний ставленням включення з кількома варіантами використання, що включаються;
- один варіант використання може бути включений до інших варіантів використання;
- в одній діаграмі варіантів використання не може бути замкнутого шляху стосовно включення.

Відношення розширення (extend) – показує, що варіант використання розширює базову послідовність дій та вставляє власну послідовність. У цьому на

відміну типу відносин «включення» розширена послідовність може здійснюватися залежно від певних умов.

Графічне зображення відношення розширення – пунктирна стрілка спрямована від залежного варіанта (розширює) до незалежного варіанта (базового) з ключовим словом <<extend>>.

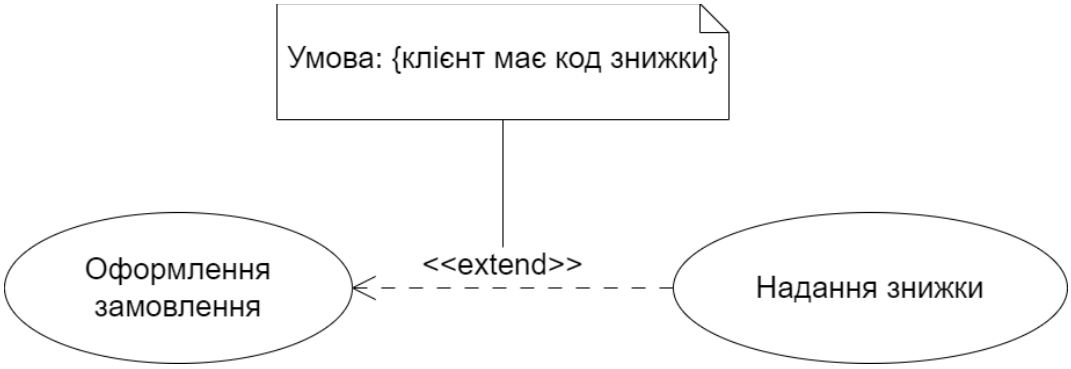

Рисунок 2.9. Приклад відношення розширення

Варіант використання «Оформлення замовлення» є базовим і може бути розширений варіантом використання "Надання знижки", наприклад, за наявності у покупця коду на знижку.

Відношення розширення дозволяють моделювати той факт, що базовий варіант використання може приєднувати до своєї поведінки деякі додаткові поведінки за рахунок розширення у варіанті іншому варіанті використання

Наявність такого відношення передбачає перевірку умови у точці розширення (extension point) у базовому варіанті використання. Точка розширення може мати деяке ім'я і зображена за допомогою примітки.

Відношення розширення з точкою розширення Особливості використання відношення розширення:

- один базовий варіант використання може мати кілька точок розширення, з кожною з яких повинен бути пов'язаний варіант використання, що розширює;

- один розширюючий варіант використання може бути пов'язаний з відношенням розширення з декількома базовими варіантами використання;
- розширюючий варіант використання може, своєю чергою, мати власні розширюючі варіанти використання;
- на одній діаграмі варіантів використання не може бути замкнутого шляху щодо розширення.

2.2.6. Сценарії використання

Діаграма варіантів використання надає знання про необхідну функціональність системи в інтуїтивно-зрозумілому вигляді, проте не несе відомостей про фактичний спосіб її реалізації. Конкретні варіанти використання можуть звучати надто загально та не бути придатними для реалізації програмістами.

Для документації варіантів використання у вигляді певної специфікації та усунення неточностей і непорозуміння діаграм варіантів використання, як частину процесу збору та аналізу вимог складаються звані сценарії використання.

Сценарії використання – це текстові уявлення тих процесів, які відбуваються при взаємодії користувачів системи та самої системи. Вони є чітко формалізованими, покроковими інструкціями, що описують той чи інший процес у термінах кроків досягнення мети. Сценарії використання однозначно визначають кінцевий результат.

Сценарії використання описують варіанти використання природною мовою. Вони мають загального, шаблонного вигляду написання, проте рекомендується такий перелік для опису:

- Передумови – умови, які повинні бути виконані для виконання даного варіанту використання;

- Постумови – що виходить в результаті виконання даного варіанту використання;
- Взаємодіючі сторони;
- Короткий опис;
- Основний перебіг подій;
- Винятки;
- Примітки.

Приклад

Розглянемо приклад входу студента до електронного кампуса (варіант використання «отримання електронних матеріалів з кампуса», один із підпроцесів).

Передумови: Немає.

Постумови: У разі успішного виконання, користувач входить до системи. У протилежному випадку стан системи не змінюється.

Взаємодіючі сторони: Студент, Кампус.

Короткий опис: Цей варіант використання визначає вхід користувача до системи реєстрації курсів.

Основний потік подій.

Цей варіант використання починає виконуватися, коли користувач хоче увійти до системи реєстрації курсів.

1. Система запитує ім'я користувача та пароль.

2. Користувач вводить ім'я та пароль.

3. Система перевіряє ім'я та пароль, після чого відкривається доступ до системи. Якщо ім'я та пароль неправильні, Виняток №1.

Винятки

Виняток №1: Неправильне ім'я/пароль. Якщо під час виконання Основного потоку виявиться, що користувач ввів неправильне ім'я та пароль, система виводить повідомлення про помилку. Користувач може повернутися до початку

Основного потоку або відмовитись від входу в систему, при цьому виконання варіанта використання завершується.

Примітки Відсутні.

2.2.7. Діаграми класів

Діаграми класів використовуються при моделюванні програмних систем найчастіше. Вони є однією із форм статичного опису системи з погляду її проєктування, показуючи її структуру [3]. Діаграма класів не відображає динамічної поведінки об'єктів зображених на ній класів. На діаграмах класів показуються класи, інтерфейси та відносини між ними.

Представлення класів

Клас – це основний будівельний блок програмної системи. Це поняття є і в мовах програмування, тобто між класами UML та програмними класами є відповідність, що є основою для автоматичної генерації програмних кодів або для виконання реінжинірингу. Кожен клас має назву, атрибути та операції. Клас на діаграмі показується як прямокутник, розділений на 3 області. У верхній міститься назва класу, у середній – опис атрибутів (властивостей), у нижній – назви операцій – послуг, що надаються об'єктами цього класу.

Атрибути класу визначають склад та структуру даних, що зберігаються в об'єктах цього класу. Кожен атрибут має ім'я та тип, який визначає, які дані він представляє.

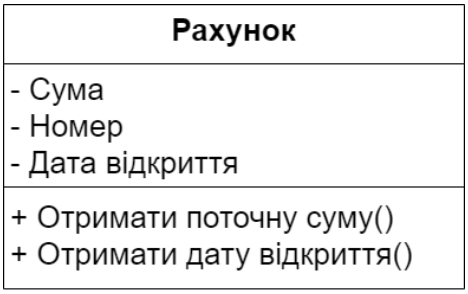

Рисунок 2.10. Зображення класу в нотації UML

При реалізації об'єкта в програмному коді для атрибутів буде виділено пам'ять, необхідна зберігання всіх атрибутів, і кожен атрибут матиме конкретне значення у час роботи програми. Об'єктів одного класу у програмі може бути скільки завгодно багато, всі вони мають однаковий набір атрибутів, описаний у класі, але значення атрибутів у кожного об'єкта свої і можуть змінюватися в ході виконання програми.

Для кожного атрибуту класу можна встановити видимість (visibility). Ця характеристика показує, чи доступний атрибут для інших класів. У UML визначено такі рівні видимості атрибутів:

• + Відкритий (public) – атрибут видно для будь-якого іншого класу (об'єкта);

• ~ В межах пакету (package) – атрибут видно для будь-якого іншого класу, який знаходиться в цьому ж пакеті;

• # Захищений (protected) – атрибут видно для нащадків цього класу;

• - Закритий (private) – атрибут не видно зовнішніми класами (об'єктами) і може використовуватися лише об'єктом, що його містить.

Клас містить оголошення операцій, що є визначення запитів, які повинні виконувати об'єкти даного класу. Кожна операція має сигнатуру, що містить ім'я операції, тип значення, що повертається, і список параметрів, який може бути порожнім. Реалізація операції як процедури – це спосіб, що належить класу. Для операцій, як й у атрибутів класу, визначено поняття «видимість». Закриті операції є внутрішніми для об'єктів класу і недоступні з інших об'єктів. Інші утворюють інтерфейсну частину класу і є засобом інтеграції класу до програмної системи.

Відносини між класами

На діаграмах класів зазвичай показуються асоціації та узагальнення.

Кожна асоціація несе інформацію про зв'язки між об'єктами усередині програмної системи. Найчастіше використовуються бінарні асоціації, які пов'язують два класи. Асоціація може мати назву, яка має виражати суть

відображуваного зв'язку. Крім назви, асоціація може мати таку характеристику як множинність. Вона показує, скільки об'єктів кожного класу може брати участь у асоціації.

Множинність вказується у кожного кінця асоціації (полюса) і задається конкретним числом чи діапазоном чисел. Множинність, зазначена у вигляді зірочки, передбачає будь-яку кількість (у тому числі й нуль). Пов'язані між собою можуть і об'єкти одного класу, тому асоціація може пов'язувати клас із собою. Наприклад, для класу «Мешканець міста» можна ввести асоціацію «Сусідство», яке дозволить знаходити всіх сусідів конкретного мешканця.

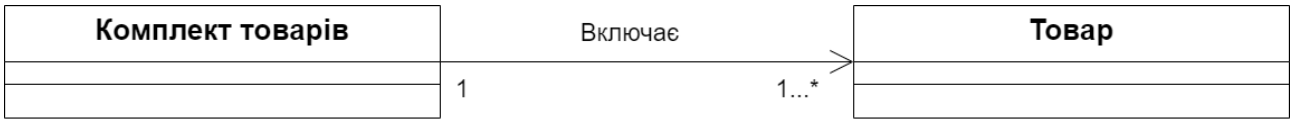

Рисунок 2.11. Асоціація між класами

Асоціація «включає» (рисунок 2.11) показує, що комплект може містити кілька різних товарів. В даному випадку спрямована асоціація дозволяє знайти всі види товарів, що входять у комплект, але не дає відповіді на запитання, чи товар цього виду входить у який-небудь набір.

Асоціація сама може мати властивості класу, тобто мати атрибути та операції. У цьому випадку вона називається клас-асоціацією і може розглядатися як клас, у якого крім явно зазначених атрибутів та операцій є посилання на обидва зв'язувані нею класи.

Асоціація – найбільш загальний вид зв'язку між двома класами системи. Як правило, вона відображає використання одного класу іншим за допомогою певної якості або поля.

Узагальнення (успадкування) на діаграмах класів використовується, щоб показати зв'язок між класом-батьком та класом-нащадком. Воно вводиться на діаграму, коли виникає різновид будь-якого класу, і навіть у випадках, як у системі виявляються кілька класів, які мають подібну поведінку (у разі загальні

елементи поведінки виносяться на більш високий рівень, утворюючи узагальнюючий клас).

Агрегацією позначається відношення частина-ціле, коли об'єкти одного класу входять до об'єкта іншого класу. Типовим прикладом таких відносин є списки об'єктів. У цьому випадку список буде виступати агрегатом, а об'єкти, що входять до списку, елементами, що агрегуються.

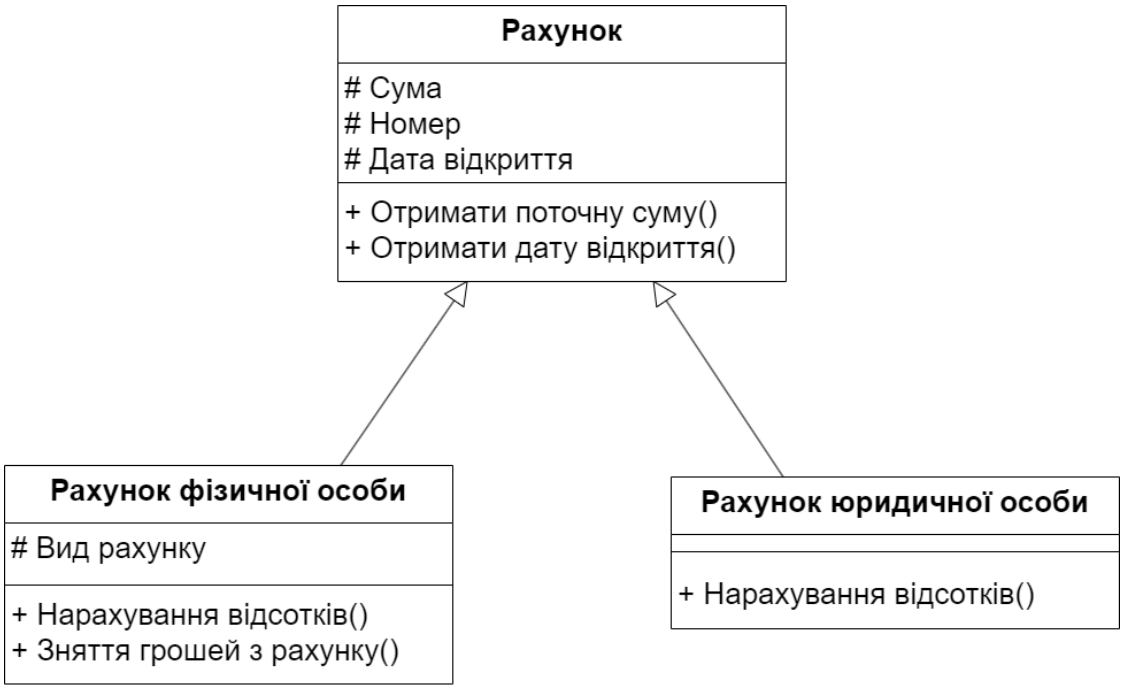

Рисунок 2.12. Приклад наслідування

Приклад агрегації показано на рисунку 2.13. Ромб в нотації частина ціле ставиться біля класу, що представляє ціле у цьому відношенні. Навчальна група містить список студентів. Композиція – це є відображенням зв’язку, при якому об’єкти «Навчальна група» та «Студент» можуть існувати один без одного. Наприклад, при видалені об’єкту «Навчальна група» об’єкти «Студент» пов’язані з нею продовжують існувати.

Композицією також позначається відношення частина-ціле, але позначає тісніший зв'язок між елементами, що представляють ціле та частини.

Рисунок 2.13. Приклад використання агрегації

Прикладом композиції можна назвати зв’язко між будівлею і приміщеннями в будівлі. Будівля та приміщення складають неділиме ціле. Приклад композиції показано на рисунку 2.14. Композиція показує, що квартири не можуть існувати за межами житлового будинку і перенести квартири до іншого житлового будинку неможливо.

Рисунок 2.14. Приклад композиції

Якщо говорити про відображення агрегації та композиції в програмному коді, то агрегація, як правило відображується як посилання на об’єкт, а композиція – це змінна типу структури. Біля цілого може бути вказана назва, яка вказує на назву поля класу, яке відображений на діаграмі цим відношенням.

Застосування діаграм класів

Діаграми класів створюються при логічному моделюванні програмних систем та служать для наступних цілей:

• Для моделювання даних. Аналіз предметної області дозволяє виявити основні характерні нею сутності та зв'язку з-поміж них. Це зручно моделюється з допомогою діаграм класів. Ці діаграми є основою побудови концептуальної моделі.

• Для представлення архітектури програмної системи. Можна виділити архітектурно значимі класи і відобразити їх на діаграмах, що описують архітектуру програмної системи.

• Для моделювання навігації екранів. На таких діаграмах показуються прикордонні класи та їхній логічний взаємозв'язок. Інформаційні поля моделюються як атрибути класів, а кнопки, що управляють, – як операції та відносини.

• Для моделювання логіки програмних компонентів.

• Для моделювання логіки обробки даних.

2.2.8. Логічна структура бази даних

Розрізняють дві моделі бази даних – логічну та фізичну. Фізична модель бази даних представляє собор набір бінарних даних у вигляді файлів, структурованих та згрупованих згідно з призначенням (сегменти, екстенти та ін.), що використовується для швидкого та ефективного отримання інформації з жорсткого диска, а також для компактного зберігання та розміщення даних на жорсткому диску.

Логічна модель бази даних є структурою таблиць, уявлень, індексів та інших логічних елементів бази даних, що дозволяють власне програмування та використання бази даних.

Процес створення логічної моделі бази даних зветься проєктування бази даних (database design). Проєктування відбувається у зв'язку з опрацюванням архітектури програмної системи, оскільки база даних створюється зберігання даних, одержуваних з програмних класів.

Відповідно можна розрізнити кілька підходів до зв'язування програмних класів та таблиць: одна таблиця – один клас, одна таблиця – кілька класів, один клас – кілька таблиць. Залежно від обраного підходу, буде ускладнюватись (і уповільнюватися) робота з базою даних при додаванні даних, або отримання даних з БД та парсинг їх у відповідні класи.

З іншого боку, якщо програмні класи є сутності проєктованої системи (елементи предметної області), то таблиці відображають їх технічну реалізацію та спосіб зберігання та зв'язку.

Основним керівництвом під час проєктування таблиць є т. зв. нормальні форми баз даних.

Нормальні форми

Нормальна форма – властивість відношення в реляційній моделі даних, що характеризує його з погляду надмірності, що потенційно призводить до логічно помилкових результатів вибірки або зміни даних. Нормальна форма окреслюється сукупністю вимог, яким має задовольняти ставлення.

Нормалізація призначена для приведення структури БД до виду, що забезпечує мінімальну логічну надмірність, і не має на меті зменшення або збільшення продуктивності роботи або зменшення або збільшення фізичного обсягу бази даних. Кінцевою метою нормалізації є зменшення потенційної суперечливості інформації, що зберігається в базі даних [4].

Змінна відношення знаходиться в першій нормальній формі (1НФ) тоді і тільки тоді, коли в будь-якому допустимому значенні відношення кожен його кортеж містить лише одне значення для кожного з атрибутів.

Змінна відношення знаходиться в другій нормальній формі тоді і тільки тоді, коли вона знаходиться в першій нормальній формі, і кожен неключовий атрибут функціонально повно залежить від її потенційного ключа.

Змінна відношення знаходиться у третій нормальній формі тоді і лише тоді, коли вона знаходиться у другій нормальній формі, і відсутні транзитивні функціональні залежності неключових атрибутів від ключових.

Змінна відношення знаходиться в нормальній формі Бойса-Кодда (інакше – в посиленій третій нормальній формі) тоді і тільки тоді, коли кожна її нетривіальна і неприведена зліва функціональна залежність має в якості свого детермінанта певний потенційний ключ.

2.2.9. Проєктування БД

Для проєктування бази даних можна використовувати Schema View у відповідних засобах роботи з СУБД (Microsoft Sql Server Management Studio, PL/SQL Developer та інші) або вбудований засіб Microsoft Office Access.

Для створення таблиць необхідно натиснути кнопку «Створити» (CREATE). Це представлено на рисунку 2.15.

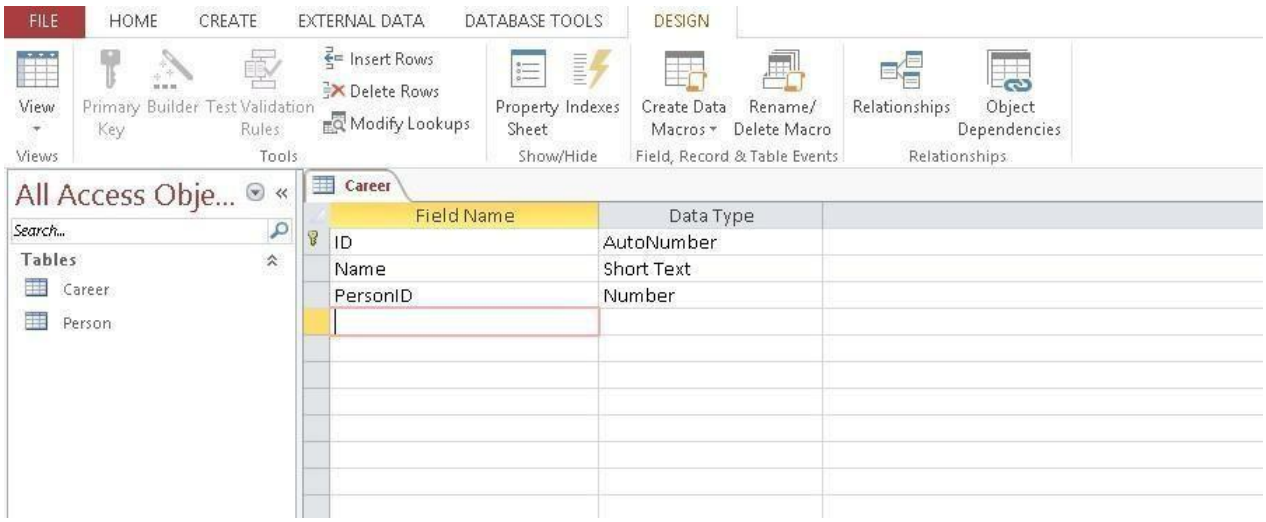

Рисунок 2.15. Створення таблиці в MS Access

Після заповнення полів та їх типів необхідно призначити первинний ключ. Після цього можна виставити зв'язок між таблицями у вікні «Зв'язки»(Relationships). Зв’язко представлено на рисунку 2.16.

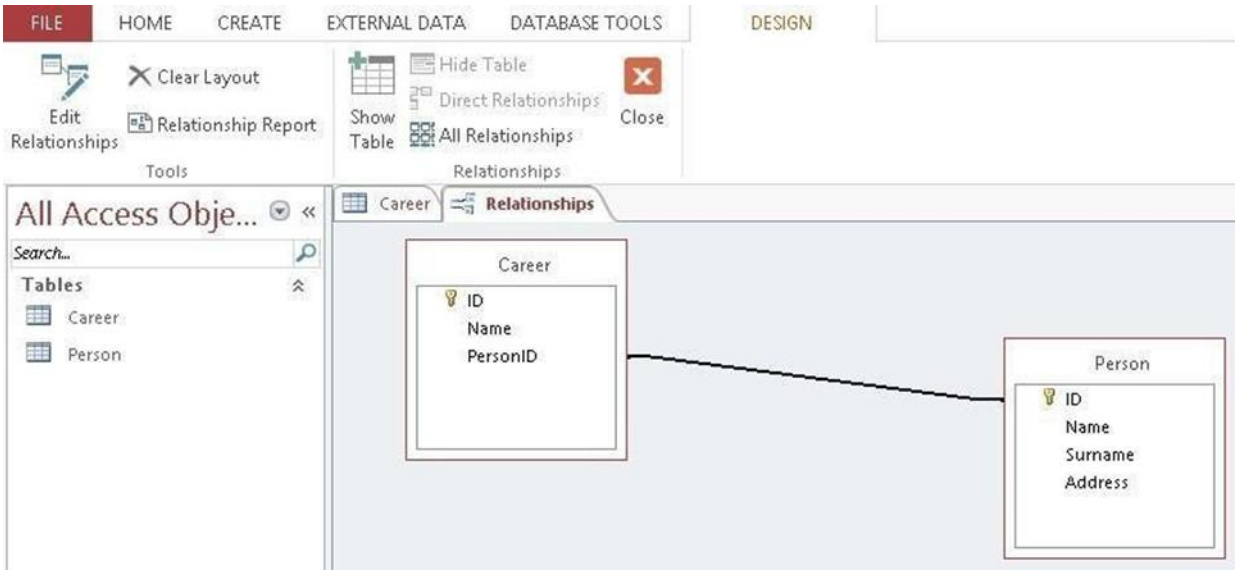

Рисунок 2.16. Зв’язок між двома таблицями на діаграмі в MS Access

2.3. Питання до лабораторної роботи

1. Що таке UML?
2. Що таке діаграма класів UML?
3. Які діаграми UML називають канонічними?
4. Що таке діаграма варіантів використання?
5. Що таке варіант використання?
6. Які відношення можуть бути відображені на діаграмі використання?
7. Що таке сценарій?
8. Що таке діаграма класів?
9. Які зв’язки між класами ви знаєте?
10. Чим відрізняється композиція від агрегації?
11. Чим відрізняється зв'язки типу агрегації від зв'язків композиції на діаграмах класів?
12. Що являють собою нормальні форми баз даних?
13. Що таке фізична модель бази даних? Логічна?
14. Який взаємозв'язок між таблицями БД та програмними класами?

3. ЛАБОРАТОРНА РОБОТА №3

Тема: Основи проектування розгортання.

Мета: Навчитися проєктувати діаграми розгортання та компонентів для системи що проєктується, а також розробляти діаграми взаємодії, а саме діаграми послідовностей, на основі сценаріїв зроблених в попередній лабораторній роботі.

3.1. Завдання

• Ознайомитись з короткими теоретичними відомостями.

• Проаналізувати діаграми створені в попередній лабораторній роботі а також тему системи та спроєктувати діаграму розгортання використання відповідно до обраної теми лабораторного циклу.

• Розробити діаграму компонентів для проєктованої системи.

• Розробити діаграму розгортання для проєктованої системи.

• Розробити як мінімум дві діаграми послідовностей для сценаріїв прописаних в попередній лабораторній роботі.

• На основі спроєктованих діаграм розгортання та компонентів доопрацювати програмну частину системи. Реалізація системи, додатково до попередньої реалізації, повинна містити як мінімум дві візуальні форми. В системі вже повинен бути повністю реалізована архітектура (повний цикл роботи з даними від вводу на формі до збереження їх в БД і подальшій виборці з БД та відображенням на UI).

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму розгортання з описом, діаграму компонентів системи з описом, діаграми послідовностей, а також вихідний код системи, який було додано в цій лабораторній роботі.

3.2. Теоретичні відомості

3.2.1. Діаграма розгортання (Deployment Diagram)

Діаграми розгортання представляють фізичне розташування системи, показуючи, на якому фізичному обладнанні запускається та чи інша складова програмного забезпечення [3].

Головними елементами діаграми є вузли, пов'язані інформаційними шляхами. Вузол (node) – це те, що може містити програмне забезпечення. Вузли бувають двох типів. Пристрій (device) – це фізичне обладнання: комп'ютер або пристрій, пов'язаний із системою. Середовище виконання (execution environment) – це програмне забезпечення, яке саме може включати інше програмне забезпечення, наприклад операційну систему або процес-контейнер (наприклад, вебсервер).

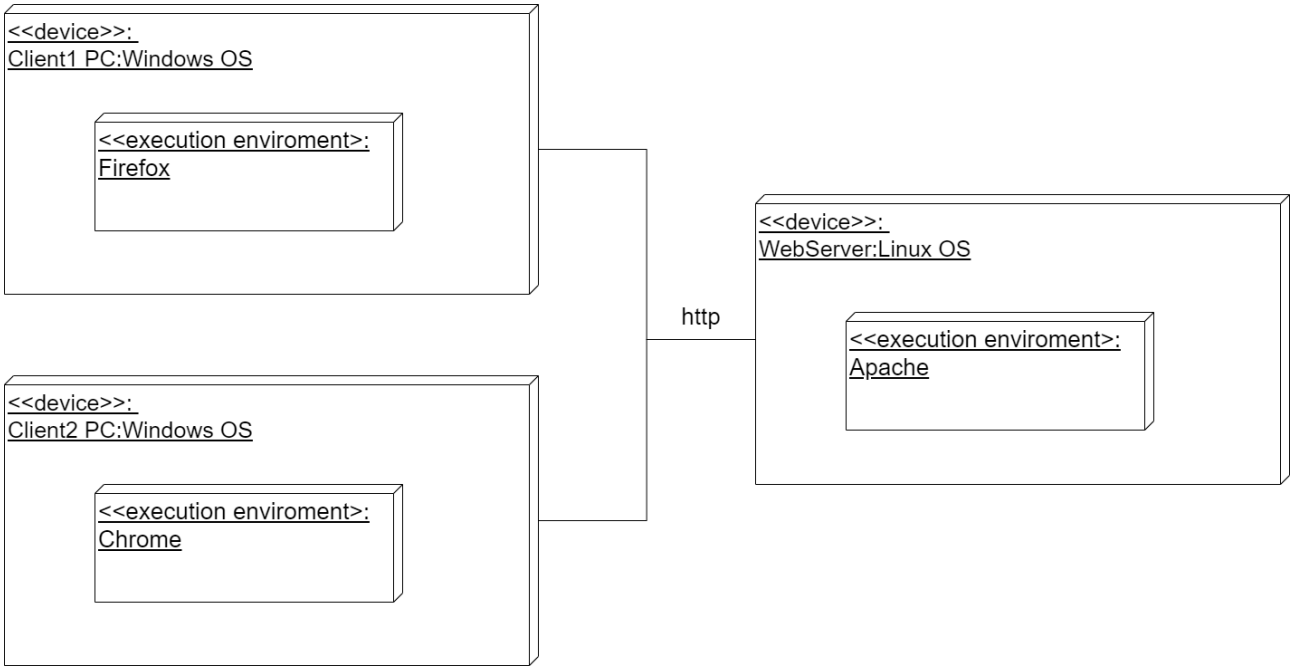

Рисунок 3.1. Діаграма розгортання системи

Між вузлами можуть стояти зв'язки, які зазвичай зображують у вигляді прямої лінії. Як і на інших діаграмах, у зв'язків можуть бути атрибути множинності (для показання, наприклад, підключення 2х і більше клієнтів до одного сервера) і назва. У назві, як правило, міститься спосіб зв'язку між двома

вузлами – це може бути назва протоколу (HTTP, IPC) або технологія, що використовується для забезпечення взаємодії вузлів (.NET Remoting, WCF).

Вузли можуть містити артефакти (artifacts), які є фізичним уособленням програмного забезпечення; зазвичай це файли. Такими файлами можуть бути виконувані файли (такі як файли .exe, двійкові файли, файли DLL, файли JAR, збірки або сценарії) або файли даних, конфігураційні файли, HTML-документи тощо. Перелік артефактів усередині вузла вказує на те, що на даному вузлі артефакт розгортається в систему, що запускається.

Артефакти можна зображати у вигляді прямокутників класів або перераховувати їхні імена всередині вузла. Якщо ви показуєте ці елементи у вигляді прямокутників класів, то можете додати значок документа або ключове слово «artifact». Можна супроводжувати вузли або артефакти значеннями у вигляді міток, щоб вказати різну цікаву інформацію про вузол, наприклад постачальника, операційну систему, місце розташування – загалом, усе, що спаде вам на думку.

Часто у вас буде безліч фізичних вузлів для розв'язання однієї й тієї самої логічної задачі. Можна відобразити цей факт, намалювавши безліч прямокутників вузлів або поставивши число у вигляді значення-мітки.

Артефакти часто є реалізацією компонентів. Це можна показати, задавши значення-мітки всередині прямокутників артефактів. Основні види артефактів:

- вихідні файли;
- виконувані файли;
- сценарії;
- таблиці баз даних;
- документи;
- результати процесу розробки, UML-моделі.

Можна також деталізувати артефакти, що входять до вузла; наприклад, додатково всередині файлу що розгортується вказати, які туди входять компоненти або класи. Така деталізація, як правило, не має сенсу на діаграмах

розгортання, оскільки може зміщувати фокус уваги від моделі розгортання програмного забезпечення до його внутрішнього устрою, проте іноді може бути корисною. При цьому, можливо встановлювати зв'язки між компонентами/класами в межах різних вузлів.

Діаграми розгортань розрізняють двох видів: описові та екземплярні. На діаграмах описової форми вказуються вузли, артефакти і зв'язки між вузлами без вказівки конкретного обладнання або програмного забезпечення, необхідного для розгортання. Такий вид діаграм корисний на ранніх етапах розроблення для розуміння, які взагалі фізичні пристрої необхідні для функціонування системи або для опису процесу розгортання в загальному ключі.

Діаграми екземплярної форми несуть у собі екземпляри обладнання, артефактів і зв'язків між ними. Під екземплярами розуміють конкретні елементи – ПК із відповідним набором характеристик і встановленим ПЗ; цілком може бути, у межах однієї організації це може бути якийсь конкретний вузол (наприклад, ПК тестувальника Василя). Діаграми екземплярної форми розробляють на завершальних стадіях розроблення ПЗ – коли вже відомі та сформульовані вимоги до програмного комплексу, обладнання закуплено і все готово до розгортання. Діаграми такої форми являють собою скоріше план розгортання в графічному вигляді, ніж модель розгортання.

3.2.2. Діаграма компонентів

Діаграма компонентів UML є представленням проєктованої системи, розбитої на окремі модулі [3]. Залежно від способу поділу на модулі розрізняють три види діаграм компонентів:

- логічні;
- фізичні;
- виконувані.

Коли використовують логічне розбиття на компоненти, то у такому разі проєктовану систему віртуально уявляють як набір самостійних, автономних модулів (компонентів), що взаємодіють між собою.

Коли на діаграмі представляють фізичне розбиття, то в такому разі на діаграмі компонентів показують компоненти та залежності між ними. Залежності показують, що класи в з одного компонента використовують класи з іншого компонента. Фізична модель використовується для розуміння які компоненти повинні бути зібрані в інсталяційний пакет. Також така діаграма показує зміни в якому компоненті будуть впливати на інші компоненти. Приклад такої діаграми наведено на рисунку 3.2.

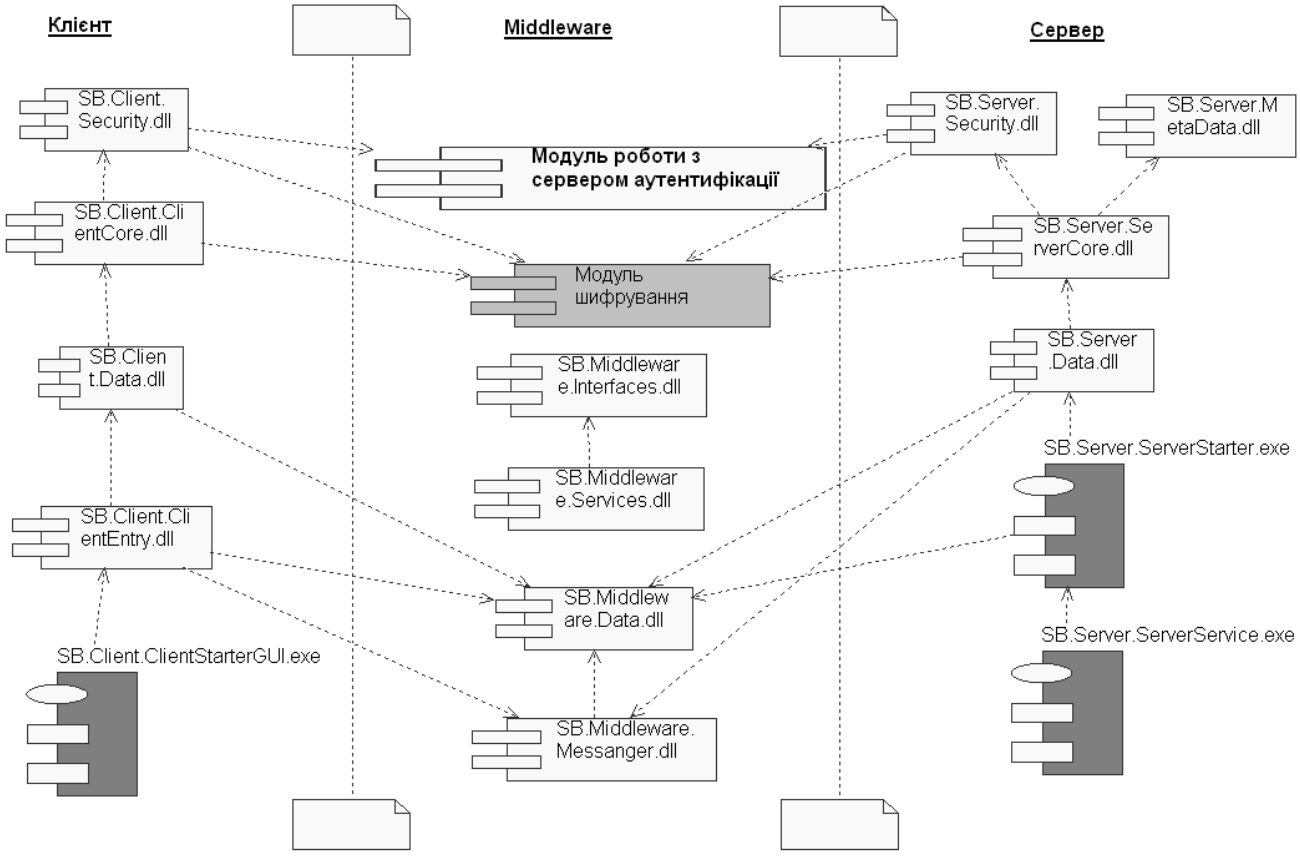

Рисунок 3.2. Приклад діаграми компонентів

У цьому випадку на діаграмі показани фізичні файли (.exe та .dll), показана залежність між ними, а також всі файли робиті на три блоки: компоненти для серверної частини, компоненти для клієнтської та компоненти, умовно названі middleware, які повинні бути як на серверній так і на клієнтській стороні, тому

що в них або є загальна бізнес-логіка, або є загальні інтерфейси методів та даних якими обмінюються між собою кліент та сервер.

Компоненти можуть поділятися за фізичними одиницями – окремі вузли розподіленої системи – набір комп'ютерів і серверів; на кожному з вузлів можуть бути встановлені різні виконувані компоненти. Такий вид діаграм компонентів застарів і зазвичай замість нього використовують діаграму розгортань.

На діаграмах компонентів з виконуваним поділом компонентів кожен компонент являє собою деякий файл – виконувані файли (.exe), файли вихідних кодів, сторінки html, бази даних і таблиці тощо.

У цьому разі діаграма схожа на діаграму класів, але на більш верхньому рівні – рівні виконуваних файлів або процесів. Такий підхід дає інший розріз представлення системи.

Діаграма компонентів розробляється для таких цілей:

- візуалізації загальної структури вихідного коду програмної системи;
- специфікації виконуваного варіанта програмної системи;
- забезпечення багаторазового використання окремих фрагментів програмного коду;
- представлення концептуальної та фізичної схем баз даних.

3.2.3. Діаграми послідовностей

Діаграма послідовностей (Sequence Diagram) – це один із типів діаграм у моделюванні UML (Unified Modeling Language), який використовується для моделювання взаємодії між об'єктами системи у певній послідовності часу. Вона відображає, як об'єкти обмінюються повідомленнями, показуючи порядок і логіку виконання операцій.

Діаграма складається з таких основних елементів:

Актори (Actors): Зазвичай позначаються піктограмами або назвами. Це користувачі чи інші системи, які взаємодіють із системою. Актори можуть бути зовнішніми стосовно моделювання системи.

Об'єкти або класи: Розміщуються горизонтально на діаграмі. Вони позначаються прямокутниками з іменем об'єкта або класу під прямокутником. Кожен об'єкт має «життєвий цикл», який представлений вертикальною пунктирною лінією (лінія життя).

Повідомлення: Це лінії зі стрілками, які з'єднують об'єкти. Вони показують передачу повідомлень чи виклик методів. Стрілка може бути синхронною (звичайна стрілка) або асинхронною (лінія з відкритим трикутником) та з пунктирною лінією, що показує повернення результату.

Активності: Вказують періоди, протягом яких об'єкт виконує певну дію. На діаграмі це позначається прямокутником, накладеним на лінію життя.

Контрольні структури: Використовуються для відображення умов, циклів або альтернативних сценаріїв. Наприклад, блоки "alt" (альтернатива) або "loop" (цикл).

Приклад діаграми послідовності зображений на рисунку 3.3.

Рисунок 3.3. Діаграма послідовності «Запит остатку на рахунку»

Основні кроки створення діаграми послідовностей:

- визначити акторів і об'єкти, які беруть участь у сценарії;
- побудувати їхні лінії життя;
- розробити послідовність передачі повідомлень між об'єктами;
- додати умовні блоки або цикли за необхідності.

Діаграми послідовностей є корисними для моделювання бізнес-процесів, проєктування архітектури систем і тестування. Вони дають змогу візуалізувати логіку взаємодії компонентів та виявити потенційні проблеми ще на етапі проєктування.

3.3. Питання до лабораторної роботи

1. Що собою становить діаграма розгортання?
2. Які бувають види вузлів на діаграмі розгортання?
3. Які бувають зв’язки на діаграмі розгортання?
4. Які елементи присутні на діаграмі компонентів?
5. Що становлять собою зв’язки на діаграмі компонентів?
6. Які бувають види діаграм взаємодії?
7. Для чого призначена діаграма послідовностей?
8. Які ключові елементи можуть бути на діаграмі послідовностей?
9. Як діаграми послідовностей пов’язані з діаграмами варіантів використання?
10. Як діаграми послідовностей пов’язані з діаграмами класів?

4. ЛАБОРАТОРНА РОБОТА №4

Тема: Вступ до паттернів проектування.

Мета: Вивчити структуру шаблонів «Singleton», «Iterator», «Proxy», «State», «Strategy» та навчитися застосовувати їх в реалізації програмної системи.

4.1. Завдання

• Ознайомитись з короткими теоретичними відомостями.

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

• Реалізувати один з розглянутих шаблонів за обраною темою.

• Реалізувати не менше 3-х класів відповідно до обраної теми.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону.

4.2. Теоретичні відомості

4.2.1. Поняття шаблону проєктування

Будь-який патерн проєктування, використовуваний при розробці інформаційних систем, являє собою формалізований опис, який часто зустрічається в завданнях проєктування, вдале рішення даної задачі, а також рекомендації по застосуванню цього рішення в різних ситуаціях [5]. Крім того, патерн проєктування обов’язково має загальновживане найменування. Правильно сформульований патерн проєктування дозволяє, відшукавши одного разу вдале рішення, користуватися ним знову і знову. Варто підкреслити, що важливим початковим етапом при роботі з патернами є адекватне моделювання розглянутої предметної області. Це є необхідним як для отримання належним чином формалізованої постановки задачі, так і для вибору відповідних патернів проєктування.

Відповідне використання патернів проєктування дає розробнику ряд незаперечних переваг. Наведемо деякі з них. Модель системи, побудована в межах патернів проєктування, фактично є структурованим виокремленням тих елементів і зв’язків, які значимі при вирішенні поставленого завдання. Крім цього, модель, побудована з використанням патернів проєктування, більш проста і наочна у вивченні, ніж стандартна модель. Проте, не дивлячись на простоту і наочність, вона дозволяє глибоко і всебічно опрацювати архітектуру розроблюваної системи з використанням спеціальної мови.

Застосування патернів проєктування підвищує стійкість системи до зміни вимог та спрощує неминуче подальше доопрацювання системи. Крім того, важко переоцінити роль використання патернів при інтеграції інформаційних систем організації. Також слід зазначити, що сукупність патернів проєктування, по суті, являє собою єдиний словник проєктування, який, будучи уніфікованим засобом, незамінний для спілкування розробників один одним.

Таким чином шаблони представляють собою, підтверджені роками розробок в різних компаніях і на різних проєктах, «ескізи» архітектурних рішень, які зручно застосовувати у відповідних обставинах.

4.2.2. Шаблон «Singleton»

Призначення патерну: «Singleton» (Одинак) являє собою клас в термінах ООП, який може мати не більше одного об'єкта (звідси і назва «одинак») [6]. Насправді, кількість об'єктів можна задати (тобто не можна створити більш n об'єктів даного класу). Даний об'єкт найчастіше зберігається як статичне поле в самому класі.

Проблема: Використання одинака виправдано для наступних випадків:

- може бути не більше N фізичних об'єктів, що відображаються в певних класах;
- необхідно жорстко контролювати всі операції, що проходять через даний клас.

Одинак вирішує відразу дві проблеми, порушуючи принцип єдиної відповідальності класу.

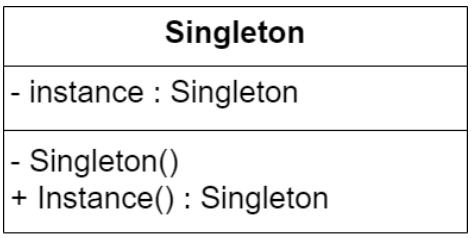

Рисунок 4.1. Структура патерну Одинак

Рішення:

Перший випадок легко продемонструвати. У кожної програми є певний набір налаштувань, який як правило зберігається в окремий файл (для сучасних комп'ютерних ігор це може бути .ini файл, для .net додатків –.xml файл). Цей файл – єдиний, і тому використання безлічі об'єктів для завантаження/запису даних – нераціональне рішення.

Для демонстрації другого випадку, розглянемо наступний приклад. Припустимо, існує дві взаємодіючі системи, між якими встановлено сеанс зв'язку. Накладені обмеження, що по даному сеансу зв'язку дані можуть йти в один момент часу лише в одну сторону. Таким чином, на кожен надісланий запит необхідно дочекатися відповіді, перш ніж відсилати новий запит. Об'єкт «одинак» дозволить не тільки містити рівно один сеанс зв'язку, а й ще реалізувати відповідну логіку перевірок на основі bool операторів про можливість відправки запиту і, можливо, деяку чергу запитів.

Переваги та недоліки: Однак слід зазначити, що в даний час патерн «Одинак» багато хто вважає т.зв. «анти-шаблоном», тобто поганою практикою проєктування. Це пов'язано з тим, що «одинаки» представляють собою глобальні дані (як глобальна змінна), що мають стан. Стан глобальних об'єктів важко відслідковувати і підтримувати коректно; також глобальні об'єкти важко тестуються і вносять складність в програмний код (у всіх ділянках коду виклик

в одне єдине місце з «одинаком»; при зміні підходу доведеться змінювати масу коду).

При цьому реалізація контролю доступу можлива за допомогою статичних змінних, замикань, мютексов та інших спеціальних структур.

+ Гарантує наявність єдиного екземпляра класу.

+ Надає до нього глобальну точку доступу.

- Порушує принцип єдиної відповідальності класу.

- Маскує поганий дизайн.

4.2.3. Шаблон «Iterator»

Призначення: «Iterator» (Ітератор) являє собою шаблон реалізації об'єкта доступу до набору (колекції, агрегату) елементів без розкриття внутрішніх механізмів реалізації. Ітератор виносить функціональність перебору колекції елементів з самої колекції, таким чином досягається розподіл обов'язків: колекція відповідає за зберігання даних, ітератор – за прохід по колекції [6].

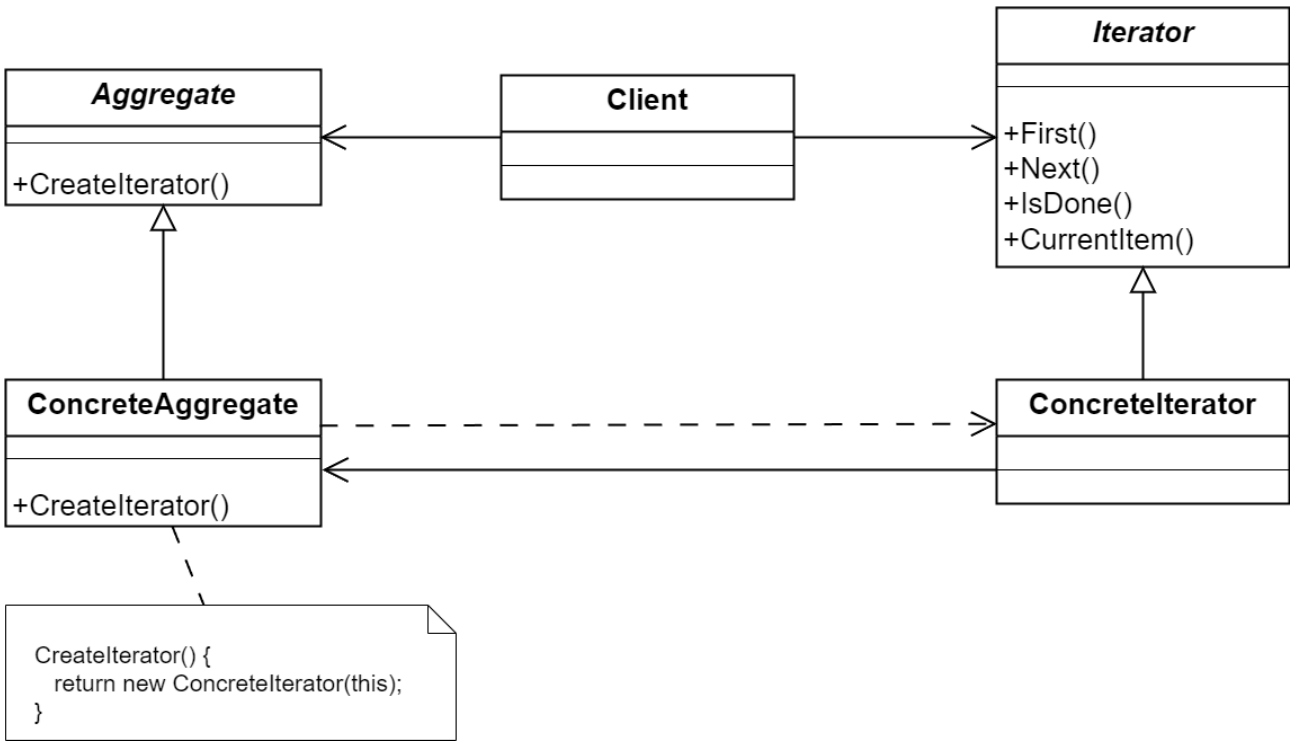

Рисунок 4.2. Структура патерну Ітератор

При цьому алгоритм ітератора може змінюватися – при необхідності пройти в зворотньому порядку використовується інший ітератор. Можливо також написання такого ітератора, який проходить список спочатку по парних позиціях (2,4,6-й елементи і т.д.), потім по непарних. Тобто, шаблон ітератор дозволяє реалізовувати різноманітні способи проходження по колекції незалежно від виду і способу представлення даних в колекції.

Проблема: Більшість колекцій виглядають як звичайний список елементів. Але є й екзотичні колекції, побудовані на основі дерев, графів та інших складних структур даних.

Але як би не була структурована колекція, користувач повинен мати можливість послідовно обходити її елементи, щоб виробляти з ними якісь дії.

Але яким способом слід переміщатися по складній структурі даних? Наприклад, сьогодні може бути достатнім обхід дерева в глибину, але завтра буде потрібно можливість переміщатися по дереву в ширину. А на наступному тижні і того гірше – знадобиться обхід колекції у випадковому порядку.

Додаючи все нові алгоритми в код колекції, ви потроху розмиваєте її основне завдання, яке полягає в ефективному зберіганні даних.

Рішення: Ідея патерна Ітератор полягає в тому, щоб винести поведінку обходу колекції з самої колекції в окремий клас.

Об'єкт-ітератор буде відстежувати стан обходу, поточну позицію в колекції і скільки елементів ще залишилося обійти. Одну і ту ж колекцію зможуть одночасно обходити різні ітератори, а сама колекція не буде навіть знати про це.

До того ж, якщо вам знадобиться додати новий спосіб обходу, ви зможете створити окремий клас ітератора, не змінюючи існуючий код колекції.

Шаблонний ітератор містить:

- First() – установка покажчика перебору на перший елемент колекції;
- Next() – установка покажчика перебору на наступний елемент колекції;
- IsDone – булевське поле, яке встановлюється як true коли покажчик перебору досяг кінця колекції;

- CurrentItem – поточний об'єкт колекції.

Переваги та недоліки: Цей шаблон дозволяє уніфікувати операції проходження по наборам об'єктів для всіх наборів. Тобто, незалежно від реалізації (масив, зв'язаний список, незв'язаний список, дерево та ін.), кожен з наборів може використовувати будь-який з реалізованих ітераторів.

+ Дозволяє реалізувати різні способи обходу структури даних.

+ Спрощує класи зберігання даних.

- Не виправданий, якщо можна обійтися простим циклом.

4.2.4. Шаблон «Proxy»

Призначення: «Proxy» (Проксі) – об'єкти є об'єктами-заглушками або двійниками/замінниками для об'єктів конкретного типу. Зазвичай, проксі об'єкти вносять додатковий функціонал або спрощують взаємодію з реальними об'єктами [5].

Проксі об'єкти використовувалися в більш ранніх версіях інтернет-браузерів, наприклад, для відображення картинки: поки картинка завантажується, користувачеві відображається «заглушка» картинки.

Проблема: Ви супроводжуєте систему, одна із частин якої працює з зовнішнім сервісом підписання документів, нарпиклад, DocuSign. Періодично клієнти в вашій системі формують звіти и форматі pdf, далі ваша система відправляє їх на підпис в сервіс DocuSign і потім періодично перевіряєте чи документ вже підписаний. Якщо документ підписаний, ви викачуєте його з сервісу і поміщаєте в своїй базі.

За останній рік кількість користувачів вашої системи виросло суттєво і почали приходити великі рахунки від DocuSign. Після аналізу ви розумієте, що рахунки зросли через велику кількість запитів з відправкою документів на підписання та запитів на перевірку чи документ вже підписаний. Після обговорення з бізнес-аналітиками та користувачам зрозуміли, що критичний інтервал доставки документів на підписання – 2 години і такий самий час критичності перевірки що документ підписаний і можна було б групувати всі

запит на відправку та на отримання і відправляти пакетом раз на годину. На даний момент клієнтський код працює з класом DocSignManager через інтерфейс IDocSignManager.

Рішення: Для вирішення проблеми можна застосувати патерн "Замісник". Ви реалізовуєте клас замісник, який також реалізовує інтерфейс IDocSignManager, але він накопичує запити на відправку файлів на підписання і відправляє їх раз на годину, також він приблизно раз на годину отримує підписані документи, а на запити від клієнтів відповідає на основі інформації взятої з бази. Таким чином старий клас DocSignManager так само використовується для роботи з DocuSign сервісом, але вже набагато рідже, а клієнтський код взаємодіє з додатковим проміжним рівнем DocSignManagerProxy, хоча з точки зору клієнського коду нічого не змінилося і він працює з тим самим об'єктом IDocSignManager.

Реалізувавши такий підхід ви тепер економите 40% від попередньої вартості використання DocuSign сервіса і тепер основний вплив на вартість робить розмір файлів, що передаються на підпис, а не кількість запитів до служби.

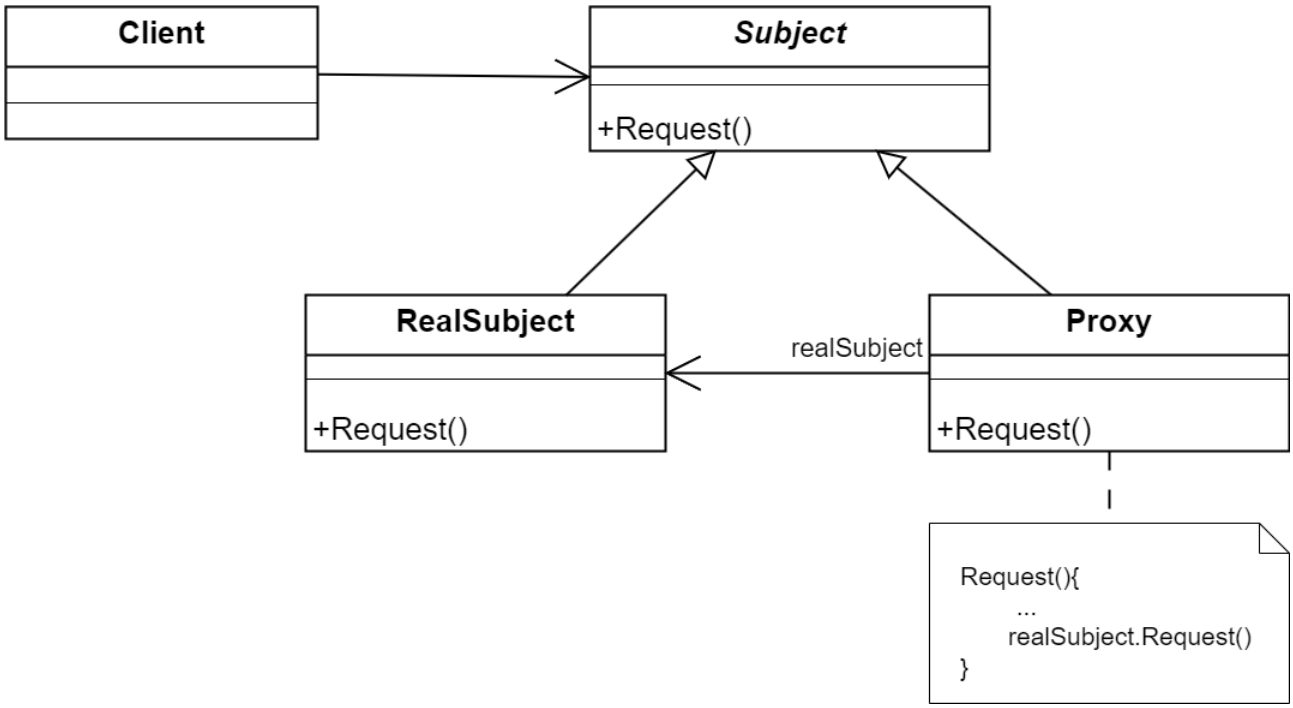

Рисунок 4.3. Структура патерну Proxy

Переваги та недоліки:

+ Легкість впровадження проміжного рівня без переробки клієнтського коду.

+ Додаткові можливості по керуванню життєвим циклом об’єкту.

- Існує ризик падіння швидкості роботи через впровадження додаткових операцій.

- Існує ризик неадекватної заміни відповіді клієнтському коду

4.2.5. Шаблон «State»

Призначення: Шаблон «State» (Стан) дозволяє змінювати логіку роботи об'єктів у випадку зміни їх внутрішнього стану [6]. Наприклад, відсоток нарахованих на картковий рахунок грошей залежить від стану картки: Visa Electron, Classic, Platinum і т.д. Або обсяг послуг, які надані хостинг компанією, змінюється в залежності від обраного тарифного плану (стану членства – бронзовий, срібний або золотий клієнт). Реалізація даного шаблону полягає в наступному: пов'язані зі станом поля, властивості, методи і дії виносяться в окремий загальний інтерфейс (State); кожен стан являє собою окремий клас (ConcreteStateA, ConcreteStateB), які реалізують загальний інтерфейс [6, 8].

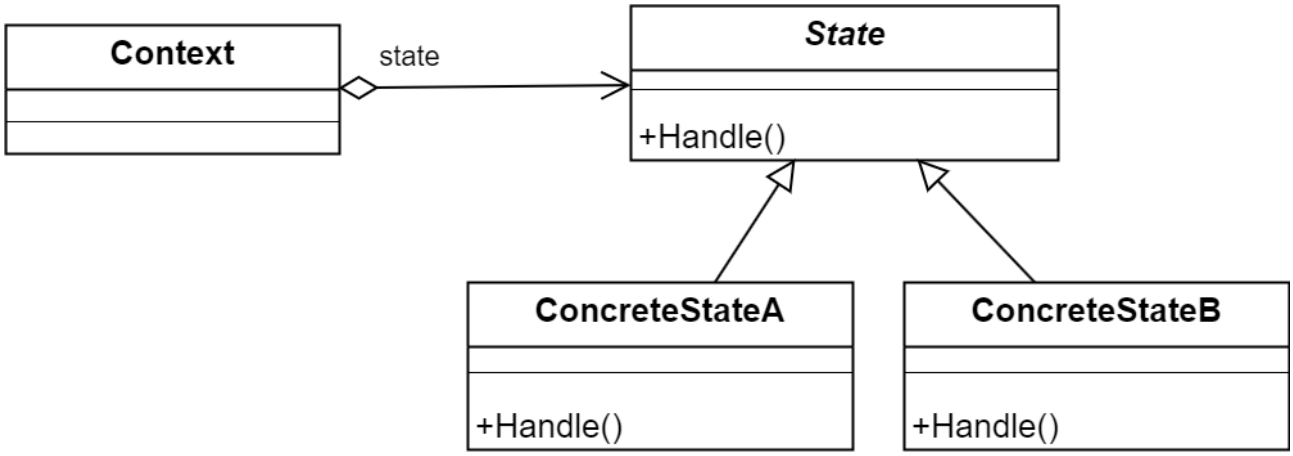

Рисунок 4.4. Структура патерну Стан

Об’єкти, що мають стан (Context), при зміні стану просто записують новий об'єкт в поле state, що призводить до повної зміни поведінки об'єкта.

Це дозволяє легко додавати в майбутньому і обробляти нові стани, відокремлювати залежні від стану елементи об'єкта в інших об'єктах, і відкрито проводити заміну стану (що має сенс у багатьох випадках).

Проблема: Ви розробляєте систему, яка складається з мобільних клієнтів та центрального сервера.

Для отримання запитів від клієнтів ви створюєте модуль Listener. Але вам потрібно щоб під час старту, поки сервер не запустився Listener не приймав запити від клієнтів, а після команди shutdown, поки сервер зупиняється, Listener вже не приймав запити від клієнтів, але відповіді, якщо вони будуть готові, відправив.

Рішення: Тут явно видно три стани для Listener з різною поведінкою: initializing, open, closing. Тому для вирішення краще за все підходить патерн State.

Визначаємо загальний інтерфейс IListenerState з методами, які по різному працюють в різних станах. Під кожний стан створюємо клас з реалізацією цього інтерфейсе. Контекст Listener при цьому всі виклики буде перенаправляти на об'єкт стану простим делегуванням. При старті він (Listener) буде посилатися на об'єкт стану InitializingState, знаходячись в якому Listener, фактично, буде ігнорувати всі вхідні запити. Після того як система запуститься і завантажить всі базові дані для роботи, Listener буде переключено в робочий стан, наприклад, викликом методу Open(). Після цього Listener буде посилатися на об'єкт OpenState і буде відпрацьовувати всі вхідні повідомлення в звичайному режимі. При запуску виключення системи, Listener буде переключено в стан ClosingState, викликом методу Close().

Переваги та недоліки:

+ Код специфічний для окремого стану реалізується в класі стану.

+ Класи та об’єкти станів можна використовувати з різними контекстами, за рахунок чого збільшується гнучкість системи.

+ Код контексту простіше читати, тому що вся залежна від станів логіка винесена в інші класи.

+ Відносно легко додавати нові стани, головне правильно змінити переходи між станами.

- Клас контекст стає складніше через ускладнений механізм переключення станів.

4.2.6. Шаблон «Strategy»

Призначення: Шаблон «Strategy» (Стратегія) дозволяє змінювати деякий алгоритм поведінки об'єкта іншим алгоритмом, що досягає ту ж мету іншим способом. Прикладом можуть служити алгоритми сортування: кожен алгоритм має власну реалізацію і визначений в окремому класі; вони можуть бути взаємозамінними в об'єкті, який їх використовує [6].

Даний шаблон дуже зручний у випадках, коли існують різні «політики» обробки даних. По суті, він дуже схожий на шаблон «State» (Стан), проте використовується в абсолютно інших цілях – незалежно від стану об'єкта відобразити різні можливі поведінки об'єкта (якими досягаються одні й ті самі або схожі цілі).

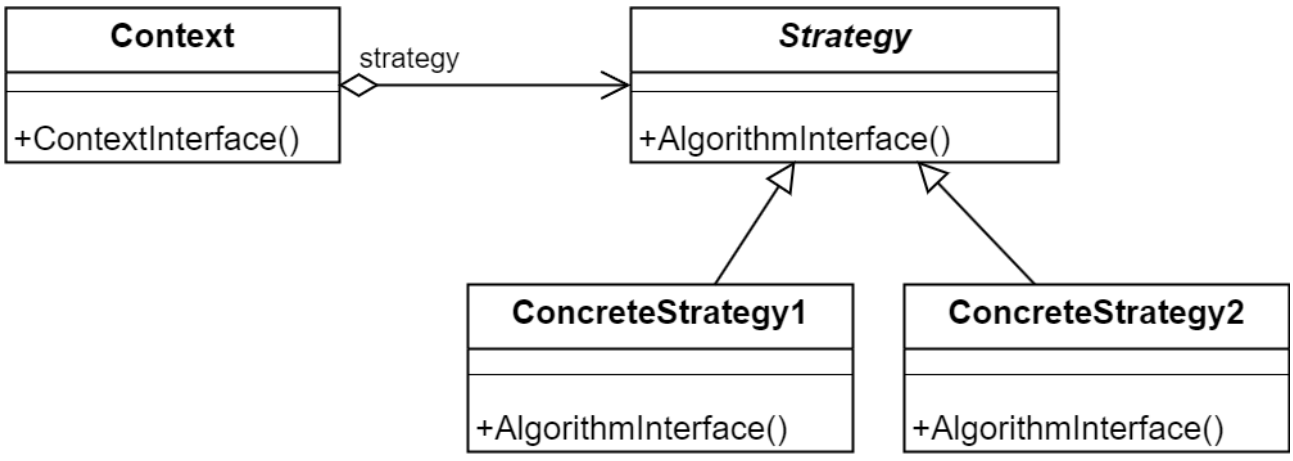

Рисунок 4.5. Структура патерну Стратегія.

Рішення: Коли ви використовуєте патерн «Стратегія», то схожі алгоритми виносяться з класа контекста в конкретні стратегії, за рахунок чого клас контексту стає чистіше і його легше супроводжувати. Також одні і тіж самі стратегії можна використати з різними контекстами, що значно збільшує гнучкість вашої системи та зменшує кількість дублювань у коді.

Контекст при цьому містить посилання на конкретну стратегію, а коли стратегію потрібно замінити, то замінюється об'єкт стратегії в полі Context.strategy.

Важливою умовою є відносна простота інтерфейсу для алгоритмів стратегій. Якщо алгоритмам стратегій прийдеться передавати десятки параметрів, то це, скоріш за все, приведе до ускладнення системи та заплутаності коду. Якщо стратегії на вході будуть приймати об'єкт контексту, щоб отримувати з нього всі необхідні дані, то такі стратегії будуть прив'язані до конкретного контексту і їх не можна буде використати з іншим типом контексту.

Приклад з життя: Ви їдете на роботу. Можна доїхати на автомобілі, на метро, або йти пішки. Тут алгоритм, як ви добираєтеся на роботу, є стратегією. В залежності від поточної ситуації ви вибираєте стратегію, що найбільше підходить в цій ситуації, наприклад, на дорогах великі пробки тоді ви їдете на метро, або метро тимчасово не ходить тоді ви їдете на таксі, або ви знаходитеся в 5 хвилинах ходьби від місця роботи і простіше добратися пішки.

Переваги та недоліки:

+ Використовувані алгоритми можна змінювати під час виконання.

+ Реалізація алгоритмів відокремлюється від коду, що його використовує.

+ Зменшує кількість умовних операторів типу switch та if в контексті.

- Надмірна складність, якщо у вас лише кілька невеликих алгоритмів.

- Під час виклику алгоритму, клієнтський код має враховувати різницю між стратегіями.

4.3. Питання до лабораторної роботи

1. Що таке шаблон проєктування?
2. Навіщо використовувати шаблони проєктування?
3. Яке призначення шаблону «Стратегія»?
4. Нарисуйте структуру шаблону «Стратегія».
5. Які класи входять в шаблон «Стратегія», та яка між ними взаємодія?
6. Яке призначення шаблону «Стан»?
7. Нарисуйте структуру шаблону «Стан».
8. Які класи входять в шаблон «Стан», та яка між ними взаємодія?
9. Яке призначення шаблону «Ітератор»?
10. Нарисуйте структуру шаблону «Ітератор».
11. Які класи входять в шаблон «Ітератор», та яка між ними взаємодія?
12. В чому полягає ідея шаблона «Одинак»?
13. Чому шаблон «Одинак» вважають «анти-шаблоном»?
14. Яке призначення шаблону «Проксі»?
15. Нарисуйте структуру шаблону «Проксі».
16. Які класи входять в шаблон «Проксі», та яка між ними взаємодія?

5. ЛАБОРАТОРНА РОБОТА №5

Тема: Патерни проектування.

Мета: Вивчити структуру шаблонів «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype» та навчитися застосовувати їх в реалізації програмної системи.

5.1. Завдання

• Ознайомитись з короткими теоретичними відомостями.

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

• Реалізувати один з розглянутих шаблонів за обраною темою.

• Реалізувати не менше 3-х класів відповідно до обраної теми.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону.

5.2. Теоретичні відомості

5.2.1. Шаблон «Adapter»

Призначення патерну: Шаблон "Adapter" (Адаптер) використовується для адаптації інтерфейсу одного об'єкту до іншого [6]. Наприклад, існує декілька бібліотек для роботи з принтерами, проте кожна має різний інтерфейс (хоча однакові можливості і призначення). Має сенс розробити уніфікований інтерфейс (сканування, асинхронне сканування, двостороннє сканування, потокове сканування і тому подібне), і реалізувати відповідні адаптери для приведення бібліотек до уніфікованого інтерфейсу. Це дозволить в програмі звертатися до загального інтерфейсу, а не приводити різні сценарії роботи залежно від способу реалізації бібліотеки. Адаптери також називаються "wrappers" (обгортками).

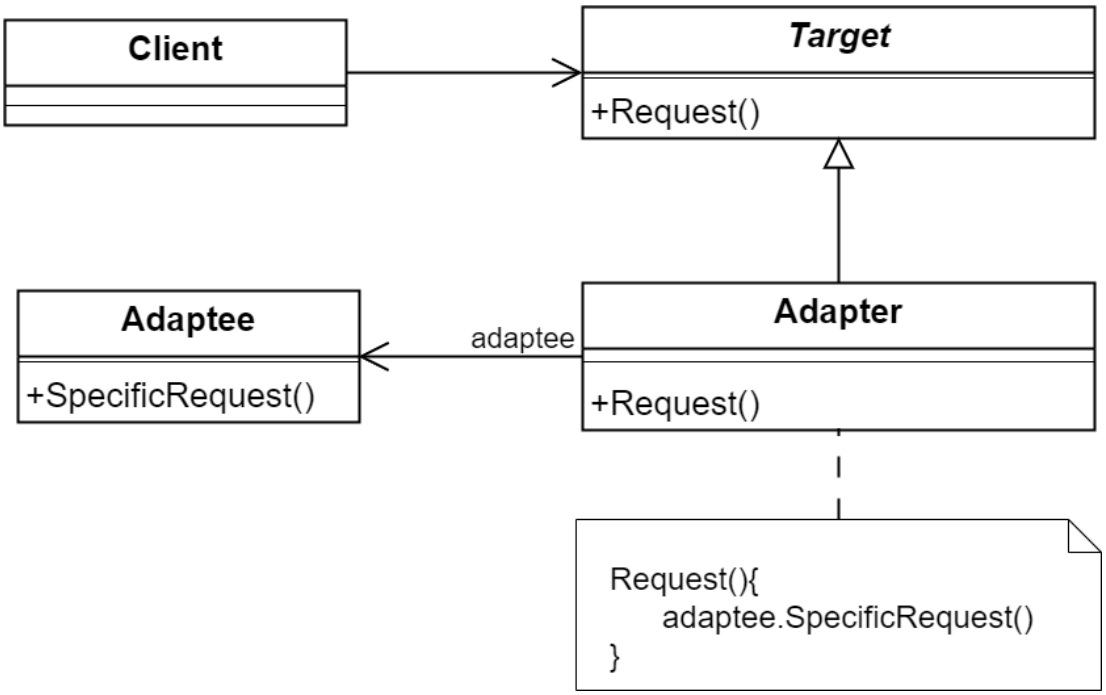

Рисунок 5.1. Структура патерну Адаптер на рівні об’єктів

Проблема: Ви реалізовуєте аудіо-плеєр, який може програвати аудіо різних форматів. Ви почали аналізувати і бачите що для програвання аудіо із різних форматів краще використовувати різні компоненти.

Таким чином ви реалізуєте складну логіку роботи з різними компонентами. У вас у коді з'являється багато логіки з перевіркою, якщо формат один, то викликаємо такий метод у такого компонента, якщо інший, то викликаємо декілька методів у іншого компонента, і т.д. Логіка роботи з компонентами стає досить складною та заплутаною.

Коли потрібно добавити підтримку нового аудіо-формату, то потрібно добавити роботу з новим компонентом і при цьому внести зміну в уже існуючу логіку, що може привести до помилок там де все коректно працювало.

Рішення: Для вирішення цих проблем можна використати патерн Адаптер. Визначаємо загальний інтерфейс IPlayer для програвання музики через компоненти. Далі для кожного компонента робимо свій адаптер. Адаптер має посилання на об'єкт компонента та реалізує інтерфейс викликаючи методи з компонента, який він адаптує. Таким чином, адаптер ніби обгортає специфічний

компонент і далі весь клієнтський код буде працювати з адаптерами через інтерфейс IPlayer. Класи, які будуть працювати з адаптером через цей інтерфейс не будуть знати інтерфейси кожного специфічного компонента.

При такій реалізації, якщо буде потрібно добавити підтримку нового формату, то просто створюється новий адаптер, який обгортає новий компонент, але робота з цим адаптером буде виконуватися через існуючий інтерфейс. При цьому існуючий код не буде зазнавати змін, а значить і не "поламаємо" те що вже працює.

Переваги та недоліки:

+ Відокремлює інтерфейс або код перетворення даних від основної бізнес-логіки.

+ Можна добавляти нові адаптери не змінюючи код у класі Client.

- Умовним недоліком можна назвати збільшення кількості класів, але за рахунок використання патерна Адаптер програмний код, як привило, стає легше читати.

5.2.2. Шаблон «Builder»

Призначення патерну: Шаблон «Builder» (Будівельник) використовується для відділення процесу створення об'єкту від його представлення [6]. Це доречно у випадках, коли об'єкт має складний процес створення (наприклад, Web-сторінка як елемент повної відповіді web- сервера) або коли об'єкт повинен мати декілька різних форм створення (наприклад, при конвертації тексту з формату у формат).

Проблема: Візьмемо процес побудови відповіді на запит web-сервера. Побудова складається з наступних частин: додавання стандартних заголовків (дата/час, ім'я сервера, інш.), код статусу (після пошуку відповідної сторінки на сервері), заголовки відповіді (тип вмісту, інш.), утримуване, інше.

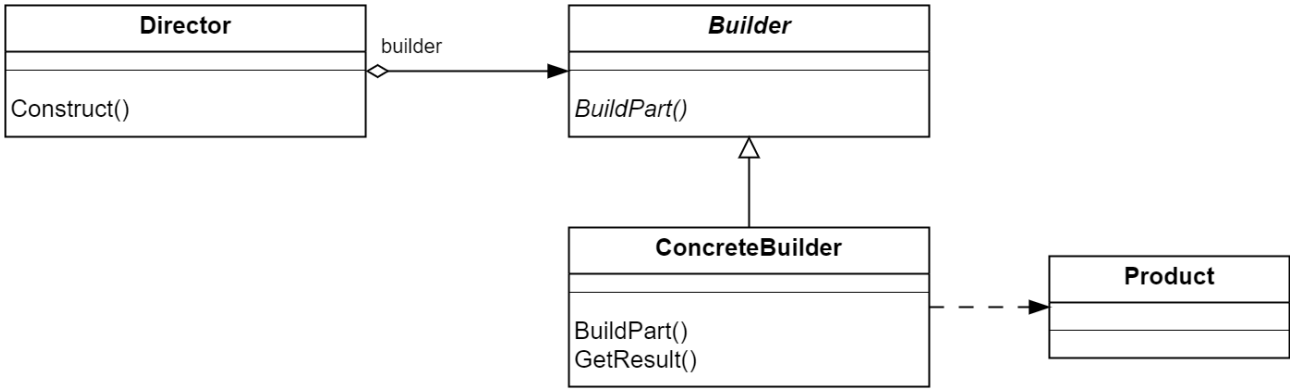

Рисунок 5.2. Структура патерну Builder

Рішення: Кожен з цих етапів може бути абстрагований в окремий метод будівельника. Це дасть наступні вигоди:

- Гнучкіший контроль над процесом створення сторінки;

- Незалежність від внутрішніх змін – наприклад, зміна назви сервера не сильно порушить процес побудови відповіді;

Переваги та недоліки:

+ Дозволяє використовувати один і той самий код для створення різноманітних продуктів.

- Клієнт буде прив’язаний до конкретних класів будівельників, тому що в інтерфейсі будівельника може не бути методу отримання результату.

5.2.3. Шаблон «Command»

Призначення патерну: Шаблон "command" (команда) перетворить звичайний виклик методу в клас [6]. Таким чином дії в системі стають повноправними об'єктами. Це зручно в наступних випадках:

- Коли потрібна розвинена система команд – відомо, що команди будуть добавлятися;
- Коли потрібна гнучка система команд – коли з'являється необхідність додавати командам можливість відміни, логування і інш.;
- Коли потрібна можливість складання ланцюжків команд або виклику команд в певний час.

Об'єкт команда сама по собі не виконує ніяких фактичних дій окрім перенаправлення запиту одержувачеві (тобто команди все ж виконуються одержувачем), однак ці об'єкти можуть зберігати дані для підтримки додаткових функцій відміни, логування і інш. Наприклад, команда вставки символу може запам'ятовувати символ, і при виклику відміни викликати відповідну функцію витирання символу. Можна також визначити параметр «застосовності» команди (наприклад, на картинці писати не можна) – і використати цей аттрибут для засвічування піктограми в меню.

Такий підхід до команд дозволяє побудувати дуже гнучку систему команд, що настроюється. У більшості додатків це буде зайвим (використовується спрощений варіант), проте життєво важливий в додатках з великою кількістю команд (редактори).

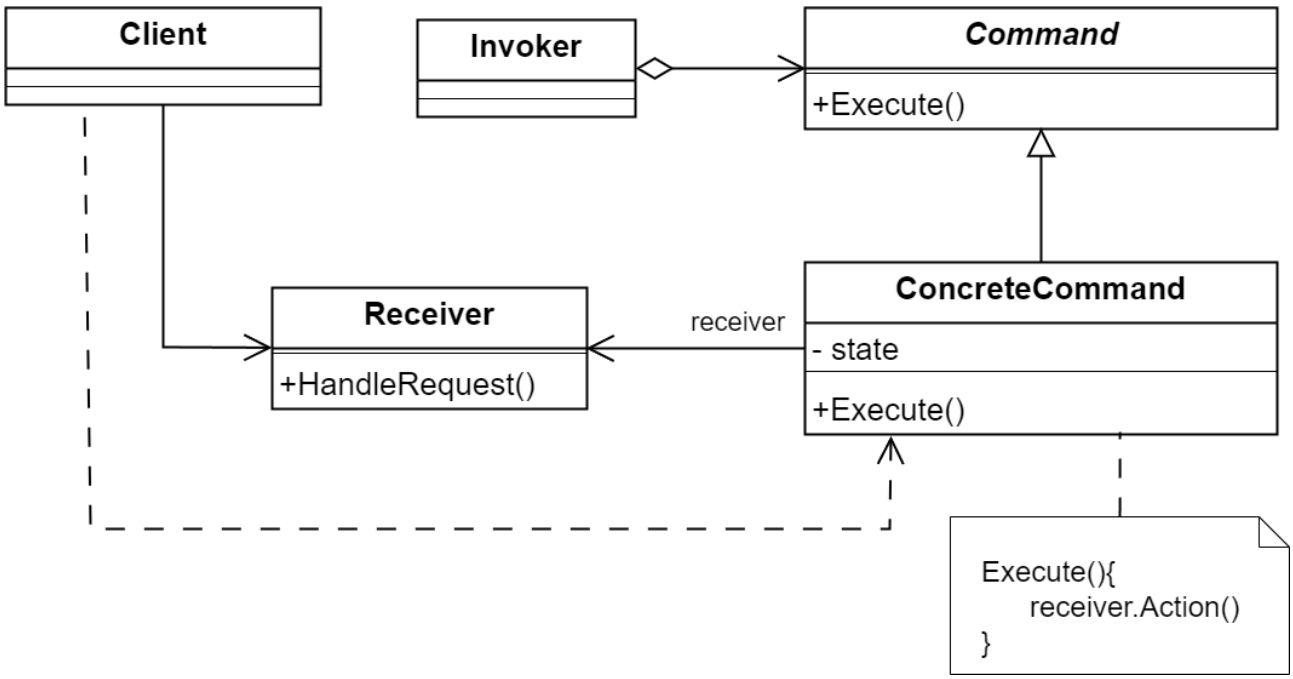

Рисунок 5.3. Структура патерну Команда

Проблема: Ви реалізуєте товстий клієнт який має багатий візуальний інтерфейс: має меню, кнопки і контекстне меню. Кожна дія, яку можна виконати, має три варіанти виконання – через меню, натисканням кнопки та через контекстне меню. Реалізацію кожної дії можна розмістити в обробнику

візуального елементу, але тоді потрібно буде продублювати цей функціонал для меню, кнопки та контекстного меню. Таким чином ми будемо мати дублювання коду і при зміні функціоналу змінювати його в трьох місцях.

Додатковим викликом буде реалізувати автоматизоване тестування такої системи, тому що нам необхідно емулювати натискання кнопок користувачем.

Рішення: Виділення функціоналу який виконується по натисканню на кнопку в окремий клас дозволяє відвязати візуальну частину від логіки обробки. Таким чином ми будемо мати шар з UI елементами і шар з логікою обробки дій виконаних на UI. Це дозволяє один і той самий об'єкт з шару логіки обробки дій використати для реакцію на натискання кнопки і пункту меню і контекстного меню. Кнопки тепер не знають про конкретні класи реалізації команд, а взаємодіють з об'єктами команд через загальний інтерфейс. Ще одна перевага, що тепер ми можемо достатньо просто реалізувати механізм enable-disable для кнопок та контекстного меню через виклик у об'єкта команди метода IsEnabled() або через прив'язку (binding) на поле IsEnabled у об'єкта команди.

При такій реалізації також набагато простіше організувати тестування застосунку, тому що ми тепер не потрібно емулювати дії користувача, а достатньо протестувати шар логіки обробки використовуючи модульні тести (unit tests).

Переваги та недоліки:

+ Ініціатор виконання команди не знає деталей реалізації виконавця команди.

+ Підтримує операції скасування та повторення команд.

+ Послідовність команд можна логувати і при необхідності виконати цю послідовність ще раз.

+ Простота розширення за рахунок додавання нових команд без необхідності внесення змін в уже існуючий код (принцип відкритості-закритості).

5.2.4. Шаблон «Chain of Responsibility»

Призначення патерну: Шаблон «Chain of responsibility» (Ланцюжок відповідальності) частково можна спостерігати в житті, коли підписання відповідного документу проходить від його складання у одного із співробітників компанії через менеджера і начальника до головного начальника, який ставить свій підпис [5].

Проблема: Ви розробляєте систему зі складними UI формами, які містять багато вкладених один в один візуальних компонентів, по кліку правою кнопкою миші вам потрібно сформувати контекстне меню. В залежності від того на якому компоненті був виконаний клік мишкою, вміст контекстного меню повинен відрізнятися, також в контекстне меню потрібно добавляти пункти, якщо компонент, в який входить поточний, теж має свої пункти. Один із очевидних підходів – ви робити метод, який викликається по кліку мишки. В методі ви робити перевірки і якщо в даний момент конкретний елемент, добавляєте в контекстне меню підходящі до нього пункти, потім перевіряєте інші елементи і перевіряєте чи вони не є такими, що містять в собі вибраний елемент. Якщо так, то добавляєте в контекстне меню ще пункти.

З часом алгоритм формування меню стає більш складним, тому що добавляються нові елементи. Також з часом в алгоритмі залишилися перевірки на компоненти, які з форми були вже видалені.

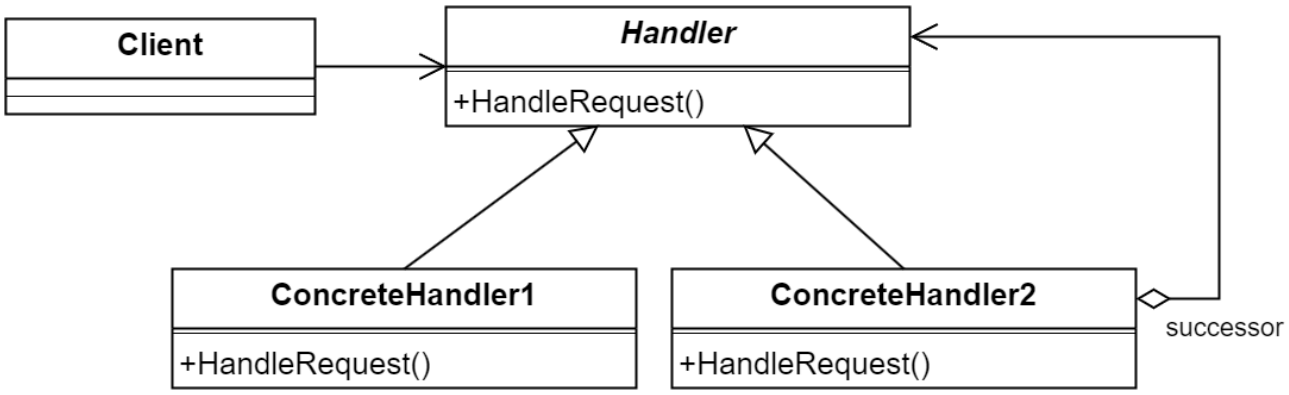

Рисунок 5.4. Структура патерна Ланцюжок відповідальності

Рішення: Для вирішення проблеми зі зростаючою складністю формування контестного меню підходить патерн «Ланцюжок відповідальності».

Основна ідея в тому, що нам не потрібно мати загальний метод формування контекстного меню. Ми можемо зробити загальний інтерфейс з методом UpdateContextMenu() і реалізувати цей метод в усіх візуальних компонентах. Додатково для візуальних компонентів добавляємо поле для переходу на "батьківський" елемент, якій в собі містить поточний візуальний компонент, а в реалізації UpdateContextMenu в кінці додаємо виклик цього метода у "батьківського" елемента. Якщо елемент не потребує додавання пунктів в контекстне меню, він просто викликає цей метод у наступного елемента в ланцюжку. Тепер обробка правого кліку мишки буде виглядати наступним чином: По кліку правою кнопкою миші викликаємо UpdateContextMenu на тексті, він, наприклад, додає пункт меню «Копіювати текст», далі викликає аналогічний метод у компонента «текст-блок» в якому цей текст знаходиться, але текст-блок нічого в контекстне меню не добавляє, а викликає UpdateContextMenu в елемента Grid, в якому знаходиться текст-блок, а Grid при відпрацюванні метода добавляє пункт меню «Контестна допомога» і так далі.

Коли ми змінюємо структуру форми, то між компонентами змінюються зв'язки в ланцюжку, але самі алгоритми вже переробляти не потрібно. якщо видаляється з форми якийсь компонент, то разом з ним видаляється і логіка пов'язана з цим компонентом. Таким чином складність роботи з контекстним меню залишається контрольованою.

Слід добавити що елемент, що обробляє виклик, не обов'язково повинен передавати виклик далі по ланцюжку. Це залежить від того, яку поведінку ви хочете отримати.

Переваги та недоліки:

+ Зменшує залежність між клієнтом та обробниками: клієнт не знає хто обробить запит, а обробники не знають структуру ланцюжка.

+ Реалізовує додаткову гнучкість в обробці запиту: легко добавити або вилучити з ланцюжка нові обробники.

- Запит може залишитися ніким не опрацьованим: запит не має вказаного обробника, тому може бути не опрацьованим.

5.2.5. Шаблон «Prototype»

Призначення патерну: Шаблон «Prototype» (Прототип) використовується для створення об'єктів за «шаблоном» (чи «кресленням», «ескізом») шляхом копіювання шаблонного об'єкту, який називається прототипом. Для цього визначається метод «клонувати» в об'єктах цього класу [6].

Цей шаблон зручно використати, коли заздалегідь відомо як виглядатиме кінцевий об'єкт (мінімізується кількість змін до об'єкту шляхом створення шаблону), а також для видалення необхідності створення об'єкту – створення відбувається за рахунок клонування, і самій програмі немає необхідності знати, як створювати об'єкт.

Також, це дозволяє маніпулювати об'єктами під час виконання програми шляхом настроювання відповідних прототипів; значно зменшується ієрархія наслідування (оскільки в іншому випадку це були б не прототипи, а вкладені класи, що наслідують).

Проблема: Ви розробляєте редактор рівнів для 2D гри на основі спрайтів. В панелі інструментів ви маєте багато кнопок для різних елементів, які можна розташовувати на екрані, такі як сходи, стіни, підлога, оздоблення та інші. Ці елементи у вас об'єднані в ієрархію з базовим класом GameObject. Під кожен елемент можна зробити свій тип кнопки, але тоді ми отримаємо паралельну ієрархію кнопок і при додаванні нового типу ігрового об’єкту потрібно буде додавати і новий тип кнопки.

Рішення: Використовуючи патерн прототип, додаємо до базового об'єкта GameObject метод Clone(), а кнопки будуть зберігати посилання на об'єкт базового типу GameObject. При натисканні на кнопку об'єкт який потрібно добавити на ігровому полі отримуємо не створенням нового, а клонуванням прототипу, який прив'язаний до кнопки. Таким чином, коли ми будемо додавати

нові типи ігрових об'єктів, то логіка роботи з кнопками не буде змінюватися, тому що не має прив'язки до конкретних типів.

Також слід відзначити, що при копіюванні об'єкту, навіть приватні поля будуть скопійовані, тому що реалізація методу копіювання знаходиться в цьому класі, що копіюється і таким чином є доступ для копіювання і до відкритих і до закритих полів.

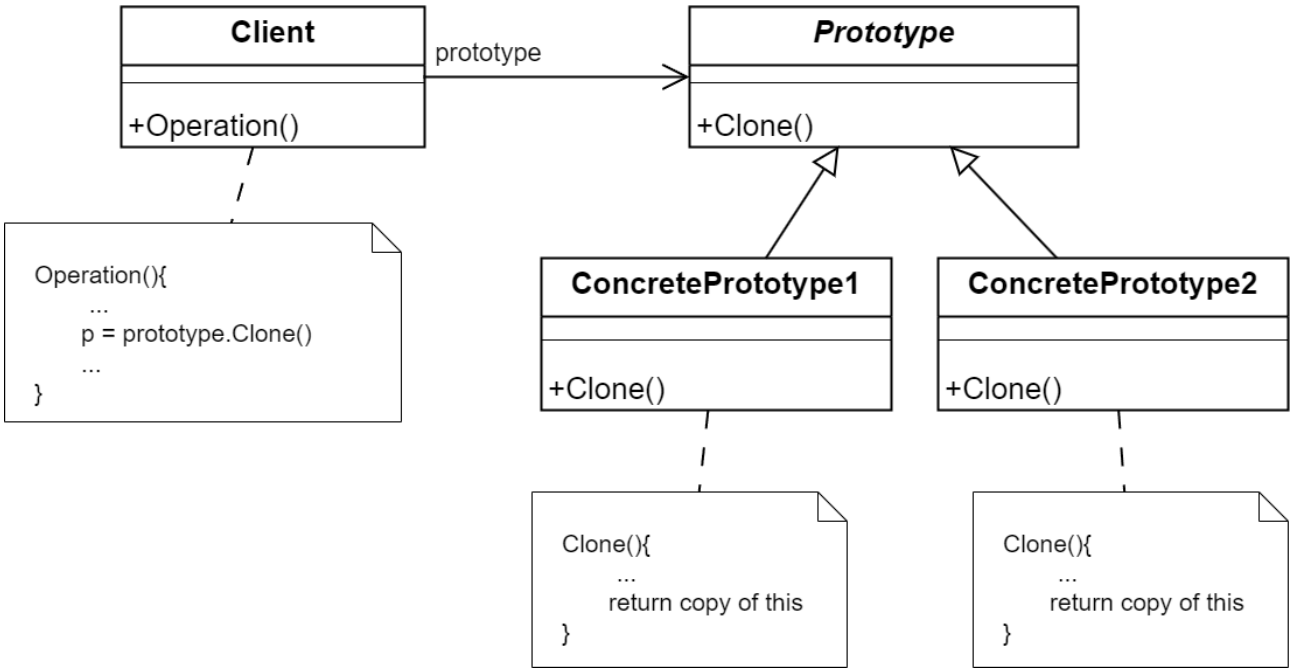

Рисунок 5.5. Структура патерну «Прототип»

Переваги та недоліки:

+ За рахунок клонування складних об’єктів замість їх створення, підвищується продуктивність.

+ Різні варіації об’єктів можна отримувати за рахунок клонування, а не розширення ієрархії класів.

+ Вища гнучкість, тому що клоновані об’єкти можна модифікувати незалежно, не впливаючи на об’єкт з якого була зроблена копія.

- Реалізація глибокого клонування досить проблематична, коли об’єкт що клонується містить складну внутрішню структуру та посилання на інші об’єкти.

- Надмірне використання патерну Прототип може привести до ускладнення коду та проблем із супроводом такого коду.

5.3. Питання до лабораторної роботи

1. Яке призначення шаблону «Адаптер»?
2. Нарисуйте структуру шаблону «Адаптер».
3. Які класи входять в шаблон «Адаптер», та яка між ними взаємодія?
4. Яка різниця між реалізацією «Адаптера» на рівні об’єктів та на рівні класів?
5. Яке призначення шаблону «Будівельник»?
6. Нарисуйте структуру шаблону «Будівельник».
7. Які класи входять в шаблон «Будівельник», та яка між ними взаємодія?
8. У яких випадках варто застосовувати шаблон «Будівельник»"?
9. Яке призначення шаблону «Команда»?
10. Нарисуйте структуру шаблону «Команда».
11. Які класи входять в шаблон «Команда», та яка між ними взаємодія?
12. Розкажіть як працює шаблон «Команда».
13. Яке призначення шаблону «Прототип»?
14. Нарисуйте структуру шаблону «Прототип».
15. Які класи входять в шаблон «Прототип», та яка між ними взаємодія?
16. Які можна привести приклади використання шаблону «Ланцюжок відповідальності»?

6. ЛАБОРАТОРНА РОБОТА №6

Тема: Патерни проектування.

Мета: Вивчити структуру шаблонів «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator» та навчитися застосовувати їх в реалізації програмної системи.

6.1. Завдання

• Ознайомитись з короткими теоретичними відомостями.

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

• Реалізувати один з розглянутих шаблонів за обраною темою.

• Реалізувати не менше 3-х класів відповідно до обраної теми.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону.

6.2. Теоретичні відомості

6.2.1. Шаблон «Abstract Factory»

Призначення патерну: Шаблон «Абстрактна фабрика» використовується для створення сімейств об'єктів без вказівки їх конкретних класів [6]. Для цього виноситься загальний інтерфейс фабрики (AbstractFactory) і створюються його реалізації для різних сімейств продуктів. Хорошим прикладом використання абстрактної фабрики є ADO.NET: існує загальний клас DbProviderFactory, здатний створювати об'єкти типів DbConnection, DbDataReader, DbAdapter та ін.; існують реалізації цих фабрик і об'єктів – SqlProviderFactory, SqlConnection, SqlDataReader, SqlAdapter і так далі. Відповідно, якщо додатку необхідно працювати з різними базами даних (чи потрібна така можливість), то досить

використати базові реалізації (Db.) і підставити відповідну фабрику у момент ініціалізації фабрики (Factory = new SqlProviderFactory()).

Цей шаблон передусім структурує знання про схожі об'єкти (що називаються сімействами, як класи для доступу до БД) і створює можливість взаємозаміни різних сімейств (робота з Oracle ведеться також, як і робота з SQL Server). Проте, при використанні такої схеми украй незручно розширювати фабрику – для додавання нового методу у фабрику необхідно додати його в усіх фабриках і створити відповідні класи, що створюються цим методом.

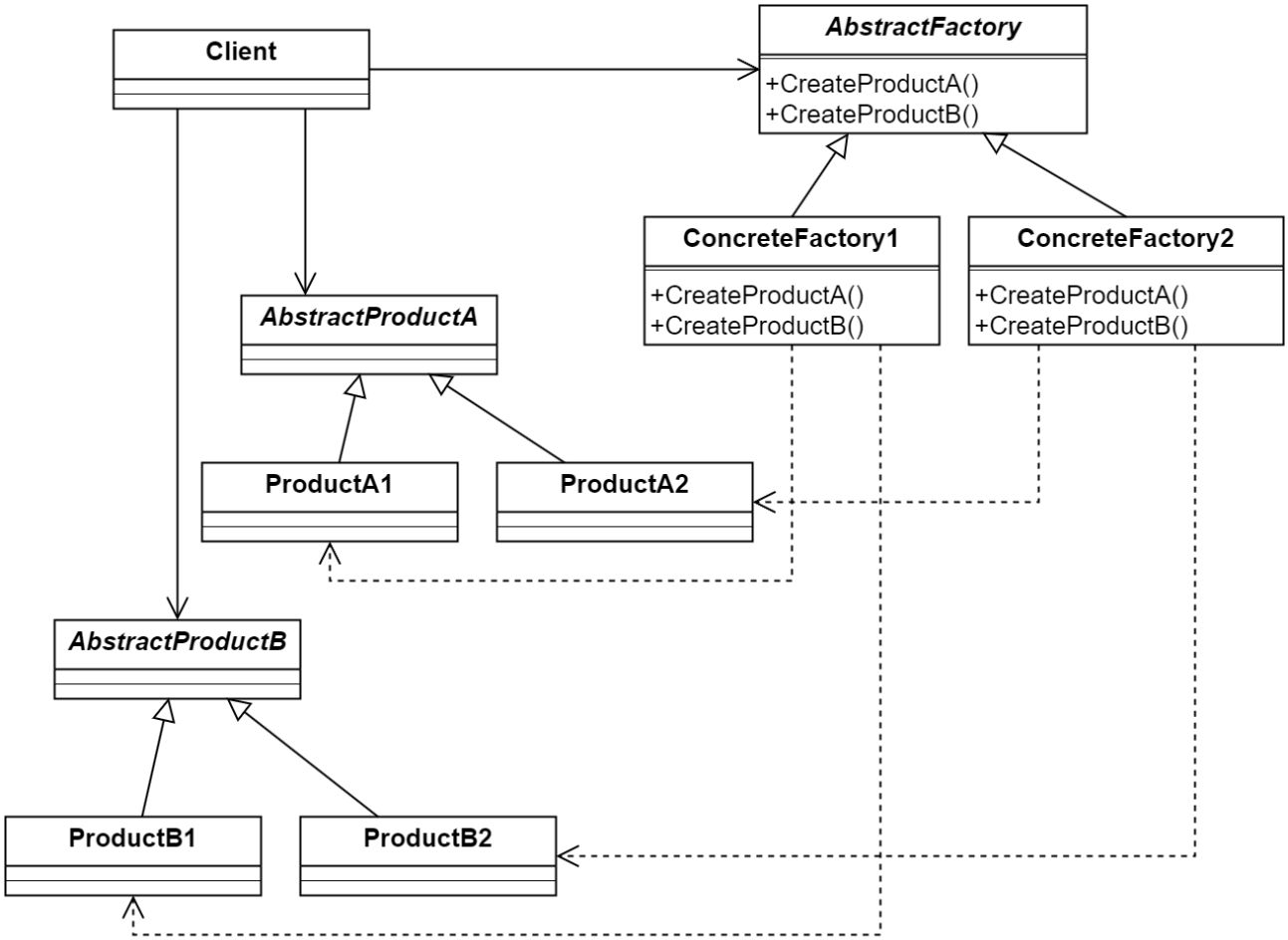

Рисунок 6.1. Структура патерну «Абстрактна фабрика»

Проблема: Ви розробляєте для Roguelike гри модуль автоматичної генерації кімнат. Модуль генерації кімнат, в процесі генерації конкретної кімнати, створює стіни, двері, вікна, підлогу та меблі в кімнаті. Модуль повинен підтримувати генерацію кімнат у різних стилях, таких як стиль хайтек, модерн,

класичний офіс. Також критично є щоб всі елементи в одній згенерованій кімнаті відносилися до одного стилю.

Рішення: Для вирішення поставленої задачі дуже добре підходить використання патерну «Абстрактна фабрика».

Для кожного типу продукту створюємо свій інтерфейс продукту який об'являє функції доступні для використання з цим продуктом. Наприклад, можна виділити інтерфейси для стін, вікон, дверей, підлоги, а також окремі інтерфейси для меблів, такі як стіл, шафа, крісло та інші. Від кожного інтерфейсу наслідуємо і реалізуємо продукти різних типів, наприклад, для стола: інтерфейс ITable та реалізації продуктів HighTechTable, ModernTable та інші.

Далі визначаємо інтерфейс для абстрактної фабрики, який буде містити методи для створення кожного типу продукту. Створюємо класи конкретних фабрик під кожен стиль: HighTechFabric, ModernFabric та інші. Кожна конкретна фабрика буде створювати всі типи продуктів, але всі вони будуть відноситися до одного стилю.

Далі в алгоритм генерації кімнати будемо передавати конкретну фабрику і алгоритм при створенні продуктів тільки через фабрику завжди буде отримувати продукти одного стилю і працювати з продуктами тільки через інтерфейс продукта.

Таким чином ми вирішуємо проблему узгодженості стилів всіх елементів в кімнаті, а за рахунок використання різних конкретних фабрик ми зможемо генерувати кімнати різних стилів. Якщо нам потрібно буде добавити ще один стиль, то достатньо буде реалізувати нові дочірні класи для кожного елемента кімнати, а також нову конкретну фабрику під цей стиль.

Переваги та недоліки:

+ Спрощує створення об’єктів і код стає легшим для розуміння.

+ Об’єкти створені однією фабрикою добре узгоджуються один з одним і зменшується кількість помилок взаємодії між ними.

+ Відокремлення створення об’єктів від їх використання, за рахунок чого, код стає біль структурованим.

+ Додавання нових сімейств продуктів виконується без зміни існуючого коду.

- Збільшується складність коду, особливо для простих проєктів.

- Додавання нового типу продукту є складним і вимагає змін коду в багатьох місцях.

6.2.2. Шаблон «Factory Method»

Призначення: Шаблон «Фабричний метод» визначає інтерфейс для створення об'єктів певного базового типу [6]. Це зручно, коли хочеться додати можливість створення об'єктів не базового типу, а деякого дочірнього. Фабричний метод у такому разі є зачіпкою для впровадження власного конструктора об'єктів. Основна ідея полягає саме в заміні об'єктів їх підтипами, що при цьому зберігає ту ж функціональність; інша частина поведінки об'єктів не є інтерфейсною (AnOperation) і дозволяє взаємодіяти із створеними об'єктами як з об'єктами базового типу. Тому шаблон «Фабричний метод» носить ще назву «Віртуальний конструктор».

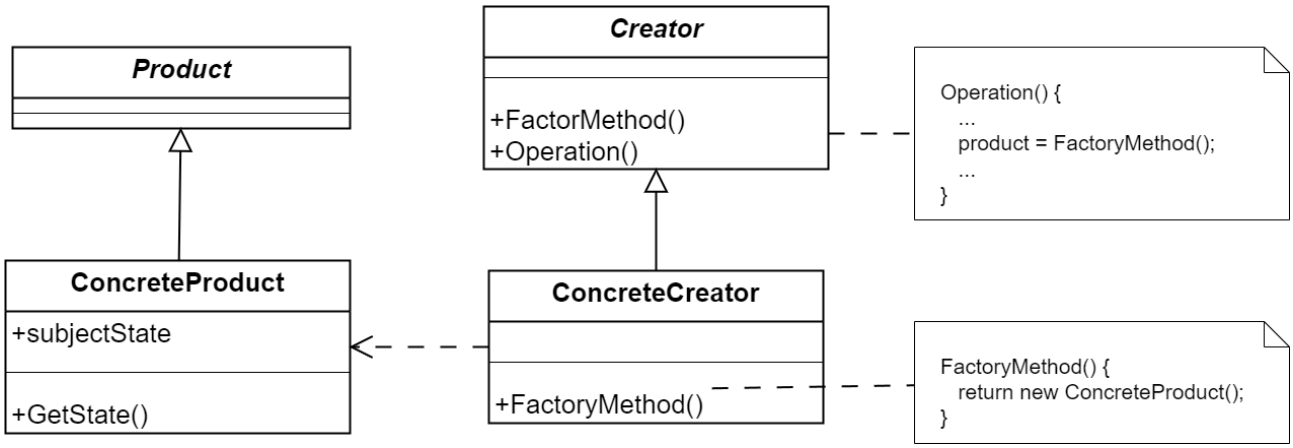

Рисунок 6.2. Структура патерну «Фабричний Метод»

Розглянемо простий приклад. Нехай наш застосунок працює з мережевими драйвер-мі і використовує клас Packet для зберігання даних, що передаються в мережу. Залежно від використовуваного протоколу, існує два перевантаження –

TcpPacket, UdpPacket. І відповідно два створюючі об'єкти (TcpCreator, UdpCreator) з фабричним методом (який створює відповідні реалізації).

Проте базова функціональність (передача пакету, прийом пакету, заповнення пакету даними) нічим не відрізняється один від одного, відповідно поміщається у базовий клас PacketCreator. Таким чином поведінка системи залишається тим же, проте з'являється можливість підстановки власних об'єктів в процес створення і роботи з пакетами.

Переваги та недоліки:

+ Позбавляє клас від прив’язки до конкретних класів продуктів.

+ Виділяє код виробництва продуктів в одне місце, спрощуючи підтримку коду.

+ Спрощує додавання нових продуктів до програми.

- Може призвести до створення великих паралельних ієрархій класів.

6.2.3. Шаблон «Memento»

Призначення: Шаблон використовується для збереження і відновлення стану об'єктів без порушення інкапсуляції [6]. Об'єкт «Memento» служить виключно для збереження змін над початковим об'єктом (Originator). Лише початковий об'єкт має можливість зберігати і отримувати стан об'єкту «Memento» для власних цілей, цей об'єкт є «порожнім» для кого-небудь ще. Об'єкт «Caretaker» використовується для передачі і зберігання мементо об'єктів в системі.

Таким чином вдається досягти наступних цілей:

- зберігання стану повністю відділяється від початкових об'єктів, що полегшує їх реалізацію;
- передача об'єктів «Memento» лягає на плечі Caretaker об'єктів, що дозволяє гнучкіше управляти станами об'єктів і спростити дизайн класів початкових об'єктів;

- збереження і відновлення стану реалізовані у вигляді двох простих методів і є закритими для кого-небудь ще окрім початкових об'єктів, таким чином не порушуючи інкапсуляцію.

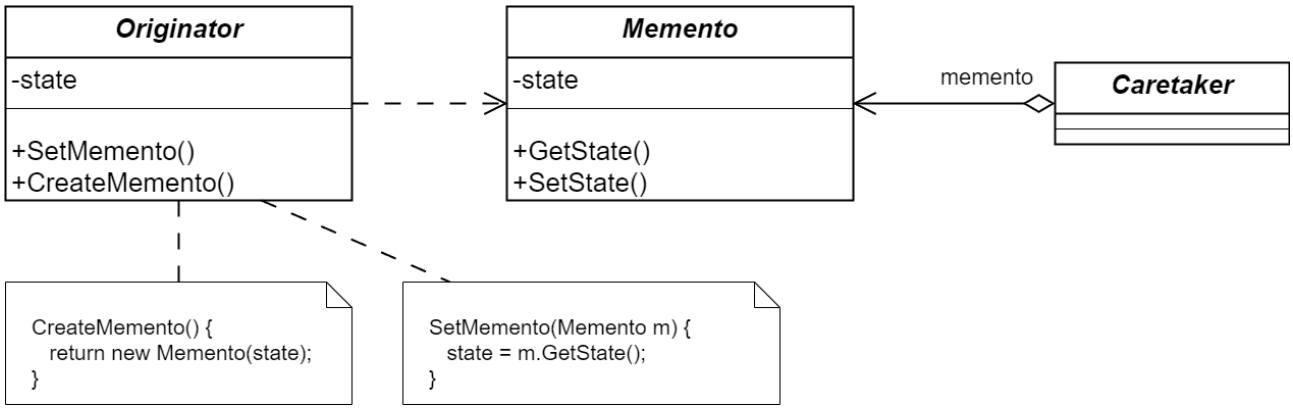

Рисунок 6.3. Структура шаблону «Знімок»

Шаблон «Мементо» дуже зручно використати разом з шаблоном «Команда» для реалізації «скасовних» дій – дані про дію зберігаються в мементо, а команда має можливість вважати і відновити початкове положення відповідних об'єктів.

Переваги та недоліки:

+ Не порушує інкапсуляцію вихідного об’єкта.

+ Спрощує структуру вихідного об’єкта. Не потрібно зберігати історію версій свого стану.

- Вимагає багато пам’яті, якщо клієнти дуже часто створюють знімки.

- Може спричинити додаткові витрати пам’яті, якщо об’єкти, що зберігають історію, не звільняють ресурси, зайняті застарілими знімками.

6.2.4. Шаблон «Observer»

Призначення: Шаблон визначає залежність «один-до-багатьох» таким чином, що коли один об'єкт змінює власний стан, усі інші об'єкти отримують про це сповіщення і мають можливість змінити власний стан також [6].

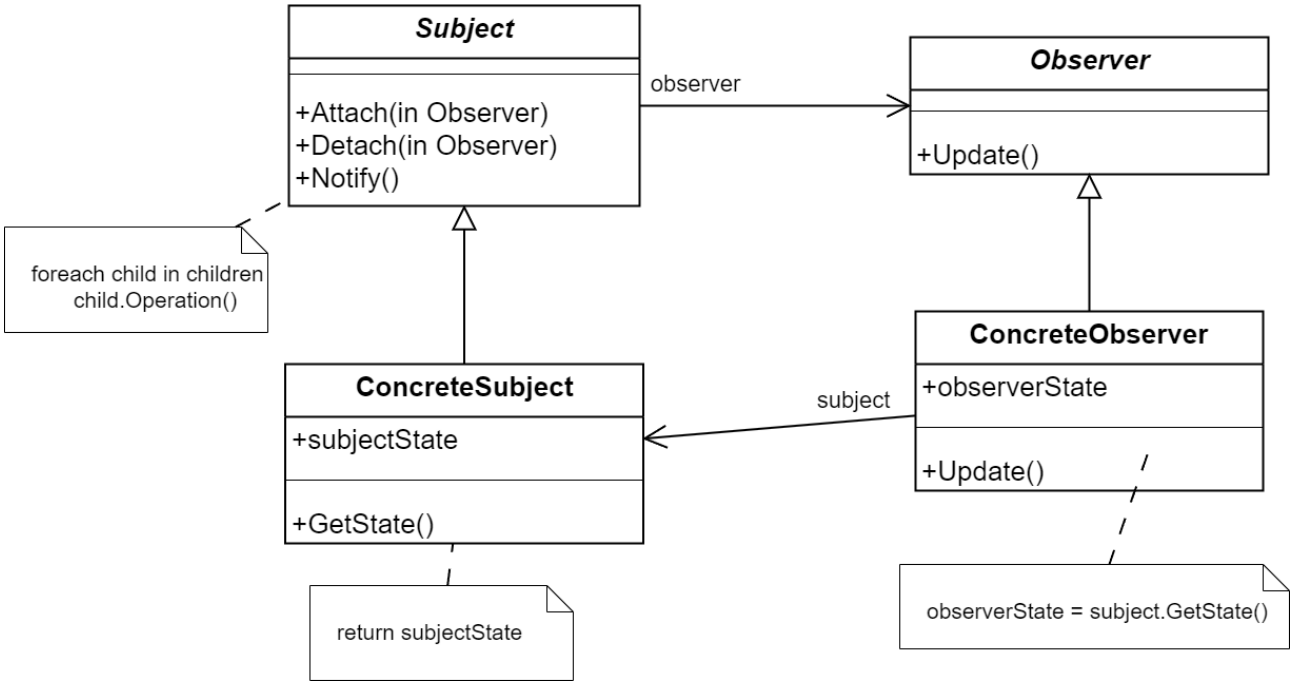

Рисунок 6.4. Структура патерна «Спостерігач»

Розглянемо цей шаблон на прикладі. Припустимо, є деяка банківська система і декілька користувачів переглядають баланс на рахунку пана И. У цей момент пан И. кладе на свій рахунок деяку суму, яка міняє загальний баланс. Кожен з користувачів, що переглядали баланс, отримує про це звістку (для користувачів ця звістка може бути прозорою – просто зміна цифр, або попередження про те, що баланс змінився). Раніше неможливі дії для користувачів (переведення до іншої категорії клієнтів і тому подібне) стають доступними.

Цей шаблон дуже широко поширений в шаблоні MVVM і механізмі «прив'язок» (bindings) в WPF і частково в WinForms. Інша назва шаблону – підписка/розсилка. Кожен з оглядачів власноручно підписується на зміни конкретного об'єкту, а об'єкти зобов'язані сповіщати своїх передплатників про усі свої зміни (на даний момент конкретних механізмів автоматичного сповіщення про зміну стану в .NET мовах не існує).

Приклад з життя: Коли ви підписуєтеся на канал на YouTube і натискаєте на «дзвіночок» ви фактично оформлюєте підписку на отримання повідомлень про вихід нових відео. Вам не потрібно заходити на канал і перевіряти чи не

вийшло нове відео. Ви будете отримувати повідомлення про вихід нових відео на каналах

Фактично на сервісі YouTube є список підписників на канал, хто має отримувати повідомлення про вихід нового відео.

В якості прикладу також можна згадати підписку «Повідомити про появу товару» на сторінці товару в магазині «Розетка».

Переваги та недоліки:

+ Можливість паралельної та асинхронної обробки повідомлень про оновлення.

+ Спостерігачів можна добавляти та видаляти в будь-який момент часу.

+ Спостерігач і суб’єкт можуть працювати в різних потоках.

+ Реалізує принцип слабкого зв’язку між об’єктами.

- Послідовність розсилки повідомлень підписникам не підтримується.

6.2.5. Шаблон «Decorator»

Призначення: Шаблон призначений для динамічного додавання функціональних можливостей об'єкту під час роботи програми [6]. Декоратор деяким чином «обертає» (за рахунок агрегації) початковий об'єкт зі збереженням його функцій, проте дозволяє додати додаткові дії. Такий шаблон надає гнучкіший спосіб зміни поведінки об'єкту чим просте спадкоємство, оскільки початкова функціональність зберігається в повному об'ємі. Більше того, таку поведінку можна застосовувати до окремих об'єктів, а не до усієї системи в цілому.

Простим прикладом є накладення смуги прокрутки до усіх візуальних елементів. Кожен об'єкт, який може прокручуватися, обертається в «прокручуваному» елементі, і при необхідності з'являється полоса прокрутки. Початкові функції елементу (наприклад, рядки статусу) залишаються незмінними.

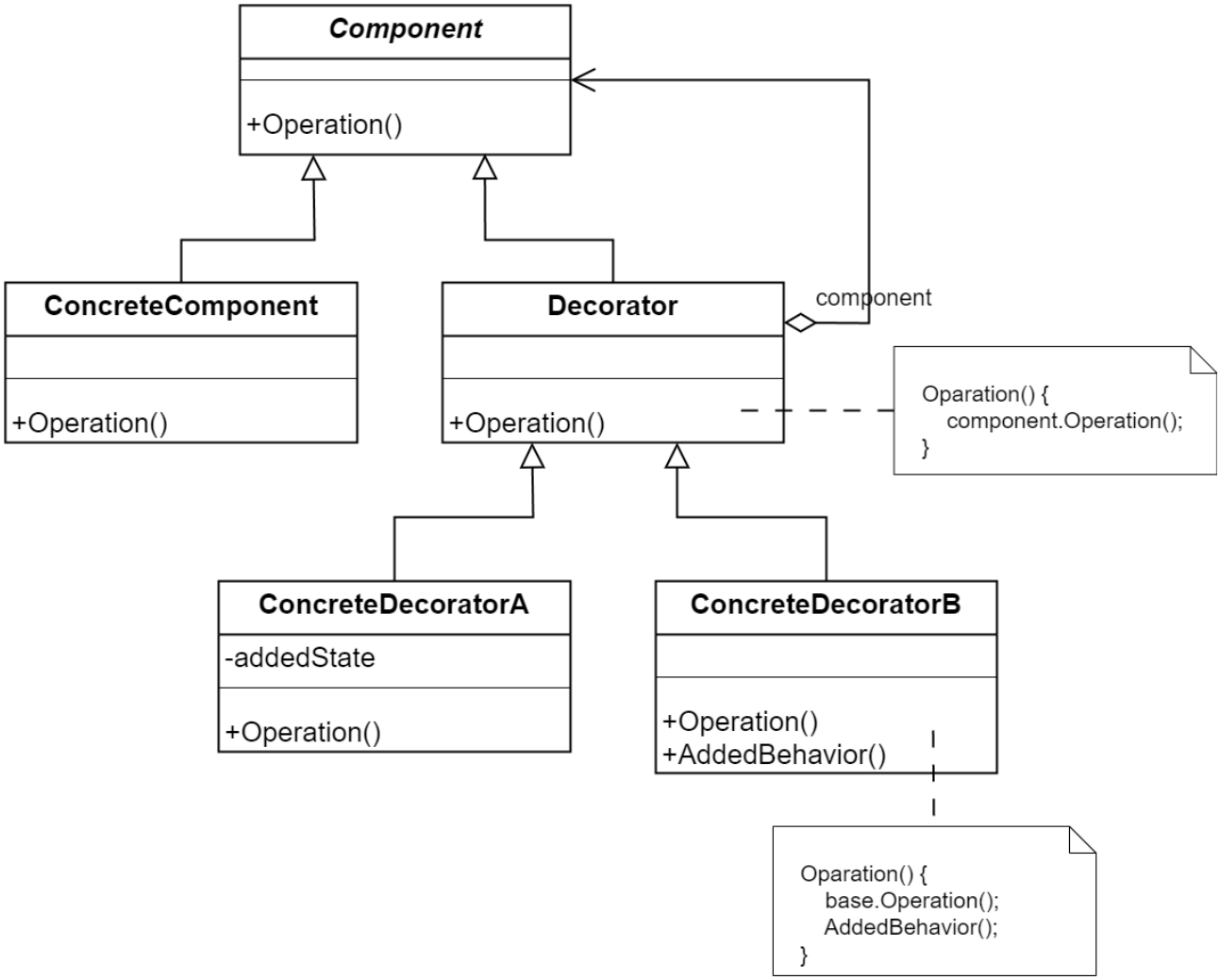

Рисунок 6.5. Структура патерну «Декоратор»

Переваги та недоліки:

+ Дозволяє мати кілька дрібних об’єктів, замість одного об’єкта «на всі випадки життя».

+ Дозволяє додавати обов’язки «на льоту».

+ Більша гнучкість, ніж у спадкування.

- Велика кількість крихітних класів.

- Важко конфігурувати об’єкти, які загорнуто в декілька обгорток одночасно.

6.3. Питання до лабораторної роботи

1. Яке призначення шаблону «Абстрактна фабрика»?

2. Нарисуйте структуру шаблону «Абстрактна фабрика».
3. Які класи входять в шаблон «Абстрактна фабрика», та яка між ними взаємодія?
4. Яке призначення шаблону «Фабричний метод»?
5. Нарисуйте структуру шаблону «Фабричний метод».
6. Які класи входять в шаблон «Фабричний метод», та яка між ними взаємодія?
7. Чим відрізняється шаблон «Абстрактна фабрика» від «Фабричний метод»?
8. Яке призначення шаблону «Знімок»?
9. Нарисуйте структуру шаблону «Знімок».
10. Які класи входять в шаблон «Знімок», та яка між ними взаємодія?
11. Яке призначення шаблону «Декоратор»?
12. Нарисуйте структуру шаблону «Декоратор».
13. Які класи входять в шаблон «Декоратор», та яка між ними взаємодія?
14. Які є обмеження використання шаблону «декоратор»?

7. ЛАБОРАТОРНА РОБОТА №7

Тема: Патерни проектування.

Мета: Вивчити структуру шаблонів «Mediator», «Facade», «Bridge», «Template method» та навчитися застосовувати їх в реалізації програмної системи.

7.1. Завдання

• Ознайомитись з короткими теоретичними відомостями.

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

• Реалізувати один з розглянутих шаблонів за обраною темою.

• Реалізувати не менше 3-х класів відповідно до обраної теми.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону.

7.2. Теоретичні відомості

7.2.1. Шаблон «Mediator»

Призначення патерну: Шаблон «Mediator» (посередник) використовується для визначення взаємодії об'єктів за допомогою іншого об'єкта (замість зберігання посилань один на одного) [6]. Даний шаблон схожий на шаблон «команда», проте в даному випадку замість зберігання даних про конкретну дію, зберігаються дані про взаємодії між компонентами.

Даний шаблон зручно застосовувати у випадках, коли безліч об'єктів взаємодіє між собою деяким структурованим чином, однак складним для розуміння. У такому випадку вся логіка взаємодії виноситься в окремий об'єкт. Кожен із взаємодіючих об'єктів зберігає посилання на об'єкт «медіатор».

«Медіатор» нагадує диригента при управлінні оркестром. Диригент стежить за тим, щоб кожен інструмент грав в правильний час і в злагоді з іншими інструментами. Функції «медіатора» повністю це повторюють.

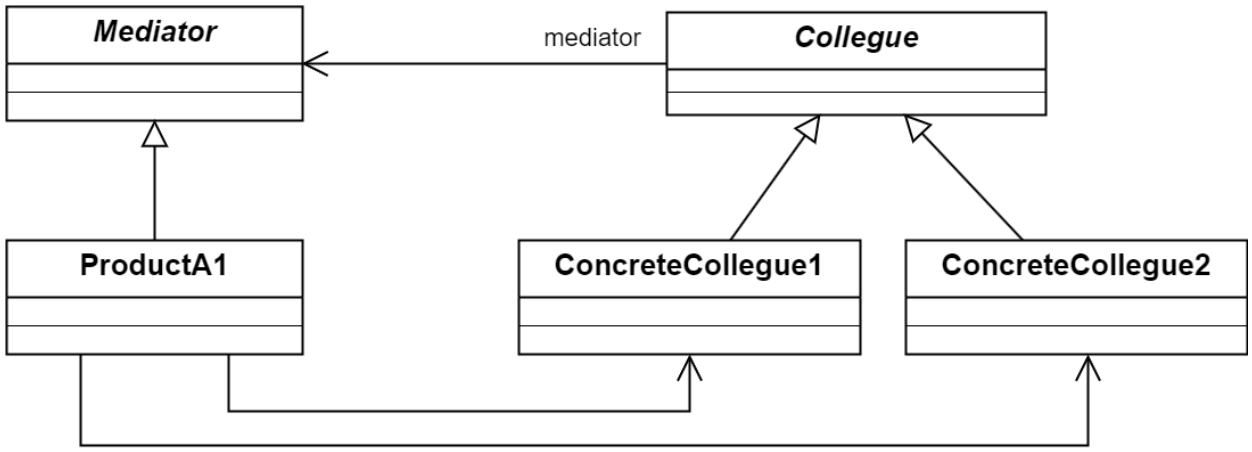

Рисунок 7.1. Структура патерна «Медіатор»

Проблема: Ви розробляєте систему зі складною візуальною частино. Головна форма з якою працює користувач складається з великою кількості візуальних компонентів, які в свою чергу складаються з елементарних візуальних компонентів. Для такої складної форми, як правило, є багато логіки, коли дії в одному компоненті повинні впливати на інші компоненти.

Наприклад, коли ви вибрали чекбокс, то після цього відключається можливість редагування даних в деяких частинах форми, деякі блоки мають приховатися, а інші навпаки, показатися. В такій ситуації, коли на формі може бути сотня, а то і більше візуальних контролів, то зв'язки між ними можуть вимірюватися тисячами. Добавляти зміни в функціонал на таких формах дуже складно, а знайти та виправити помилку, взагалі, може бути дуже затратно по витраченому часу.

Рішення: В таких ситуаціях дуже добре підходить патерн «Mediator» (Посередник). При реалізації такого патерна, ми добавляємо новий клас посередник, і всі взаємодії між компонентами повинні відбуватися вже через нього. За рахунок цього компоненти вже не знають один про одного, а лише

знають про посередника і для відпрацювання логіки звертаються вже до нього. А вже посередник взаємодіє з іншими компонентами для відпрацювання цієї логіки.

Основною перевагою, яку ми отримуємо при такому підході є сильно зменшена зв'язність між компонентами. А за рахунок цього стає набагато простіше розібратися з взаємодією між об'єктами коли потрібно добавити на форму нові об'єкти або добавити якийсь функціонал. Якщо компоненти будуть взаємодіяти не напряму з посередником, я через загальний інтерфейс посередника, то тоді можна буде підміняти посередника, наприклад, на тестового, щоб емулювати тестові випадки для перевірки.

Переваги та недоліки:

+ Організація взаємодії між об’єктами лише через посередника спрощує розуміння та супроводження такого коду.

+ Додавання нових посередників без зміни існуючих компонентів дозволяє розширювати систему без зміни існуючого коду.

+ Зменшення залежностей між об’єктами підвищує гнучкість системи.

- Посередник, з часом, може перетворитися на дуже складний об’єкт, який робить все («God Object»).

7.2.2. Шаблон «Facade»

Призначення патерну: Шаблон «Facade» (фасад) передбачає створення єдиного уніфікованого способу доступу до підсистеми без розкриття внутрішніх деталей підсистеми [6]. Оскільки підсистема може складатися з безлічі класів, а кількість її функцій – не більше десяти, то щоб уникнути створення «спагеті-коду» (коли все тісно пов'язано між собою) виділяють один загальний інтерфейс доступу, здатний правильним чином звертатися до внутрішніх деталей.

Це також відволікає користувачів від змін в підсистемі (внутрішня реалізація може змінюватися, а наданої послуги немає), що також скоротить кількість змін в використовуваних фасад класах (без фасаду довелося б змінювати вихідні коди в безлічі точок).

Звичайно, твердої умови повного закриття внутрішніх класів підсистеми не стоїть – при необхідності можна звертатися до окремих класів безпосередньо, минаючи об'єкт фасад.

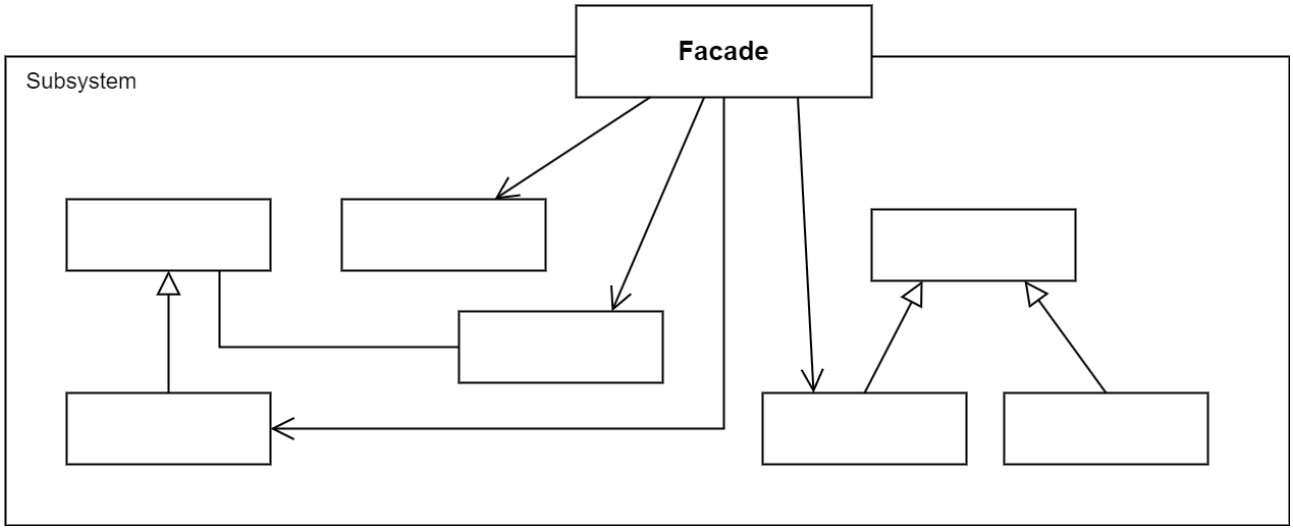

Рисунок 7.2. Структура патерну «Фасад»

Проблема: Ви розробляєте компонент, який дозволяє відправляти запити на різні типи endpoint, а також працює з протоколами HTTP та TCP/IP.

Прототип компонента вже працює, але структура класів вийшла досить складна, а при налаштуванні на різні протоколи мають використовуватися різні класи.

Інструкція для використання також виходить досить складна та заплутана. Слід добавити, так як інші системи будуть знати про внутрішню будову вашого компонента, а тому при спробі змінити внутрішню структуру в наступних версіях, вам прийдеться повідомляти про це всіх користувачів вашого компонента і для них перехід на наступну версію вашого компонента буде достатньо складним.

Рішення: Тут краще використати патерн фасад. А саме, в даному випадку, створити один клас, наприклад, InternetClient, та набір методів у нього, які будуть використовуватися для налаштування цього підключення, та його подальшого використання. Цей клас єдиний буде позначено як public, і тільки його будуть

бачити ваші клієнти. Таким чином, з точки зору зовнішнього коду вони будуть працювати з набагато простішим інтерфейсом, а значить і інструкція використання буде значно простіша. Крім того, так як внутрішня структура повністю закрита, то в наступних версіях ви можете її змінювати, як вам буде зручно, а з точки зору користувачів компонента, перехід на нову версію буде просто переключенням на нову версію бібліотеки.

Переваги та недоліки:

+ Інкапсуляція внутрішньої структури від клієнтського коду.

+ Спрощується інтерфейс для роботи з модулем закритим фасадом.

- Зниження гнучкості в налаштуванні та використанні програмного коду закритого фасадом.

7.2.3. Шаблон «Bridge»

Призначення патерну: Шаблон «Bridge» (міст) використовується для поділу інтерфейсу і його реалізації. Це необхідно у випадках, коли може існувати кілька різних абстракцій, над якими можна проводити дії різними способами.

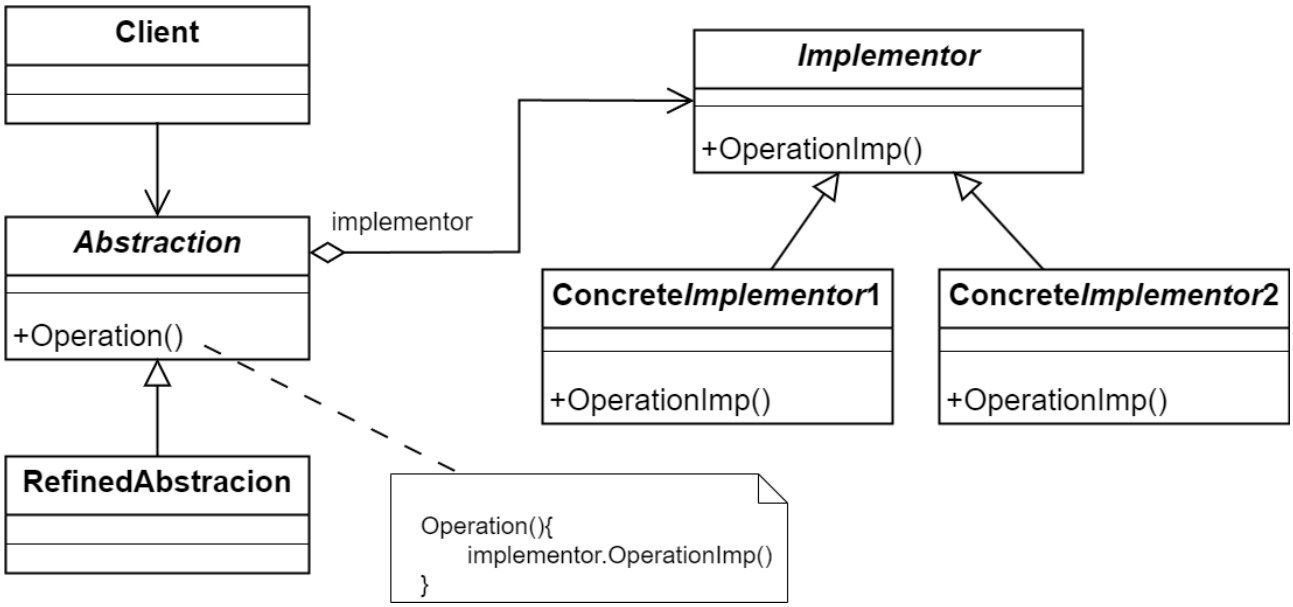

Рисунок 7.3. Структура патерну «Міст»

Проблема: Ви реалізовуєте графічний векторний редактор, який дозволяє рисувати круги, прямокутники, прямі та довільні лінії.

Ви маєте реалізувати функціонал відображення отриманого рисунку на екрані та друкувати на принтер.

Спростимо ситуацію: ваш редактор дозволяє рисувати лише лінії та круги.

При такому підході, нам потрібно будувати ієрархію фігур з різною реалізацією дочірніх класів: LinePrint, LineDraw, CirclePrint, CircleDraw.

якщо добавити прямі, то добавиться ще два підкласи, і т.д. А як бути, коли нам потрібно буде ще і реалізувати збереження в bitmap форматі? добавляємо ще LineBinery, CircleBinery? При такому підході ми отримуємо дуже складну ієрархію класів.

Рішення: В даному випадку ми можемо використати патерн «Міст» (Bridge): робимо дві ієрархії – фігур (Shape) та рисування (DrawApi).

При такому підході DrawApi – це інтерфейс імплементації відображення (графічного драйвера), а Shape – інтерфейс абстракції фігур, яка має агрегацію з об'єктом DrawApi. При такому підході фігури будуть делегувати рисування об'єкту DrawApi.

Лінія, коло, та інші будуть дочірніми класами до Shape, а WindowDrawApi та PrinterDrawApi – дочірні класи до DrawApi, які представляють графічні драйвери для відображення на екрані та принтері відповідно. Якщо нам потрібно буде добавити ще і збереження в bitmap форматі, то ми добавимо ще один підклас реалізації графічного драйвера BitmapDrawApi. Таким чином ми маємо дві різні ієрархії об’єктів і вони в нас не перетинаються і не збільшуються в геометричній прогресії при додаванні нових драйверів або фігур.

Також слід відмітити, що DrawApi нічого не знає про фігури (абстракцію), а дочірні класи абстракції не залежать від реалізації графічного драйвера.

Переваги та недоліки:

+ Дозволяє змінювати ієрархії абстракції та реалізації незалежно одна від одної.

+ Розділивши абстракцію від реалізації отримуємо більшу гнучкість та простіший супровід такого коду.

- Підвищена гнучкість при використанні патерну отримується за рахунок більшої складності, введення додаткових проміжних рівнів.

7.2.4. Шаблон «Template Method»

Призначення патерну: Шаблон «Template Method» (шаблонний метод) дозволяє реалізувати покроково алгоритм в абстрактному класі, але залишити специфіку реалізації підкласам [6]. Можна привести в приклад формування веб-сторінки: необхідно додати заголовки, вміст сторінки, файли, що додаються, і нижню частину сторінки. Код для додавання вмісту сторінки може бути абстрактним і реалізовуватися в різних класах – AspNetCompiler, HtmlCompiler, PhpCompiler і т.п. Додавання всіх інших елементів виконується за допомогою вихідного абстрактного класу з алгоритмом.

Даний шаблон дещо нагадує шаблон «Фабричний метод», однак область його використання абсолютно інша – для покрокового визначення конкретного алгоритму; більш того, даний шаблон не обов'язково створює нові об'єкти – лише визначає послідовність дій.

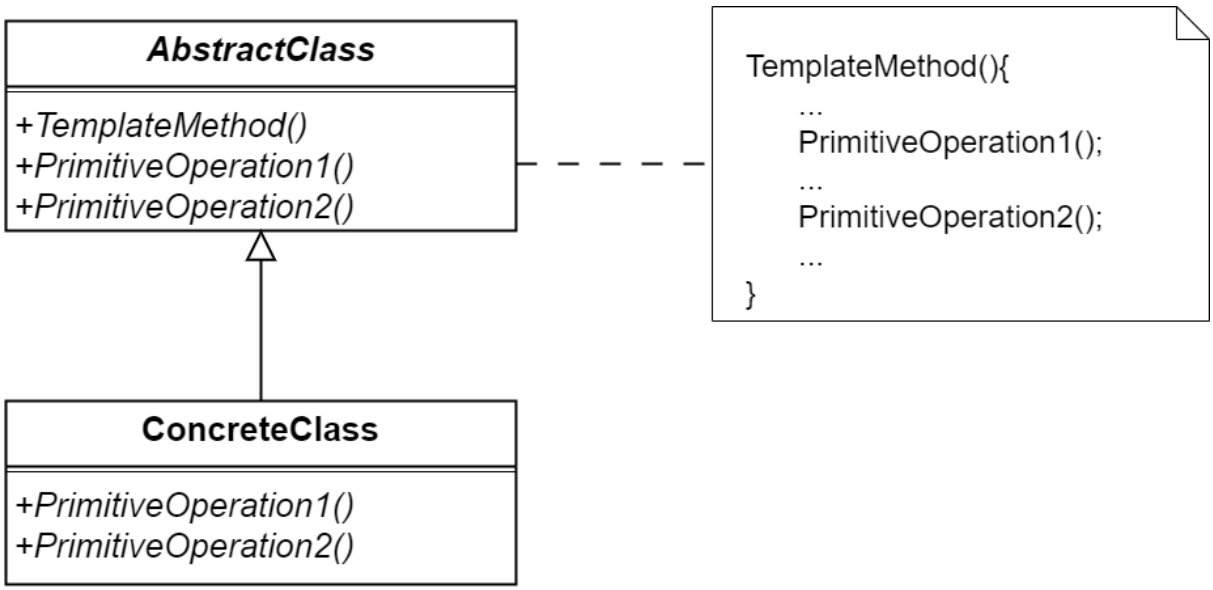

Рисунок 7.4. Структура патерну «Шаблонний метод»

Проблема: Ви працюєте в команді, що займається розробкою застосунку для редагування відео-файлів. Застосунок вже працює з форматом відео MPEG-4, а саме дозволяє читати такі файли, виконувати попередню обробку даних для відображення в відео-редакторі.

Ви отримуєте нову задачу на реалізацію можливості роботи з більш старим форматом MPEG-2. Ви бачите два варіанта: зробити копію існуючого класу, що працює з MPEG-4, або вносити зміни в уже існуючий клас. Щоб прийняти рішення ви більш детально розбираєтеся з існуючим алгоритмом і бачите, що близько 70 відсотків коду має бути таким самим. Тому ви вирішуєте змінити вже існуючий клас для роботи з MPEG-4 добавляючи в місцях де це потрібно умови з перевіркою, що якщо формат MPEG-2 то відпрацьовувати новий код, який ви добавили. Через деякий час, на запити від користувачів, вам на реалізацію приходить задача добавити підтримку ще більш старого формату MPEG-1. Ви вносите зміни так само в існуючий клас, тільки умови стали більш складними, тому що розгалуження логіки йде на три гілки.

Ще через деякий час приходить аналогічна задача на додавання читання даних з файлів формату H.262. Ви починаєте працювати над задачею і бачите, що код, який до цього був ще більш-менш зрозумілим стає зовсім важким для читання та внесення змін.

Рішення: Патерн «Шаблонний метод» (Template Method) пропонує загальний алгоритм винести в базовий клас, а частини алгоритма, які для різних задач виконуються по різному, виділити в окремі методи. Ці методи будуть викликатися в алгоритмі, що реалізований в базовому класі. В дочірніх класах ці виділені методи будуть перевизначатися. Таким чином загальна логіка залишається в базовому класі, а специфічна частина реалізується в дочірніх класах.

Якщо подивитися на задачу з відео-редактором, то застосування «Шаблонного методу» наведе лад в коді і спростить його зміни.

Як це зробити: По перше, в алгоритмі всі блоки коду де є вибір гілки на основі типу формату виділяються в окремі методи. У випадку з відео-редактором, це скоріш за все будуть блоки коду пов'язані з читанням даних та розпакування їх в кадри, а також читання звукових доріжок. Далі створюється загальний базовий клас в який переноситься загальний алгоритм, а також об'являються віртуальні методи (фактично беремо сигнатуру тих методів, що виділили на попередньому кроці). Далі створюємо дочірні класи під кожен формат файлу і перевизначаємо віртуальні методи. Фактично при цьому в кожному такому методі в дочірньому класі із реалізації цих методів, що була виділена на першому кроці, залишається код гілки який відповідав вибраному формату.

Після всіх цих змін ми маємо реалізацію патерна «Шаблоний метод»: в базовому класі реалізовано базовий алгоритм (по суті більша частина алгоритму) і в дочірніх класах перевизначені методи зі специфічною логікою.

Після таких змін, добавити підтримку нового формату стає легше, тому що достатньо буде добавити лише новий дочірній клас і перевизначити в ньому необхідні методи.

Слід зауважити, що якщо у вас алгоритми співпадають більше ніж на 50 відсотків, то застосування шаблонного методу буде доцільним, але якщо у вас алгоритми співпадають лише відсотків на 10 або 20, то скоріш за все, краще буде використати патерн «Стратегія».

Переваги та недоліки:

+ Полегшує повторне використання коду.

- Ви жорстко обмежені скелетом існуючого алгоритму.

- Ви можете порушити принцип підстановки Барбари Лісков, змінюючи базову поведінку одного з кроків алгоритму через підклас.

- З ростом складності загального алгоритму шаблонний метод стає занадто складно підтримувати, особливо, коли є багато віртуальних методів для перевизначення в підкласах.

7.3.Питання до лабораторної роботи

1. Яке призначення шаблону «Посередник»?
2. Нарисуйте структуру шаблону «Посередник».
3. Які класи входять в шаблон «Посередник», та яка між ними взаємодія?
4. Яке призначення шаблону «Фасад»?
5. Нарисуйте структуру шаблону «Фасад».
6. Які класи входять в шаблон «Фасад», та яка між ними взаємодія?
7. Яке призначення шаблону «Міст»?
8. Нарисуйте структуру шаблону «Міст».
9. Які класи входять в шаблон «Міст», та яка між ними взаємодія?
10. Яке призначення шаблону «Шаблоний метод»?
11. Нарисуйте структуру шаблону «Шаблоний метод».
12. Які класи входять в шаблон «Шаблоний метод», та яка між ними взаємодія?
13. Чим відрізняється шаблон «Шаблонний метод» від «Фабричного методу»?
14. Яку функціональність додає шаблон «Міст»?

8. ЛАБОРАТОРНА РОБОТА №8

Тема: Патерни проектування.

Мета: Вивчити структуру шаблонів «Composite», «Flyweight» (Пристосуванець), «Interpreter», «Visitor» та навчитися застосовувати їх в реалізації програмної системи.

8.1. Завдання

• Ознайомитись з короткими теоретичними відомостями.

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

• Реалізувати один з розглянутих шаблонів за обраною темою.

• Реалізувати не менше 3-х класів відповідно до обраної теми.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону.

8.2. Теоретичні відомості

8.2.1. Шаблон «Composite»

Призначення: Шаблон використовується для складання об'єктів в деревоподібну структуру для подання ієрархій типу «частина цілого». Даний шаблон дозволяє уніфіковано обробляти як поодинокі об'єкти, так і об'єкти з вкладеністю [6].

Простим прикладом може служити складання компонентів всередині звичайної форми. Форма може містити дочірні елементи (поля для введення тексту, цифр, написи, малюнки тощо); дочірні елементи можуть в свою чергу містити інші елементи. Наприклад, при виконанні операції розтягування форми необхідно, щоб вся ієрархія розтягнулася відповідним чином. В такому випадку

форма розглядається як композитний об'єкт і операція розтягування застосовується до всіх дочірніх елементів рекурсивно.

Даний шаблон зручно використовувати при необхідності подання та обробки ієрархій об'єктів. Крім того, патерн «Composite» (Компонувальник) краще використовувати, коли ви представляєте структуру даних у вигляді дерева.

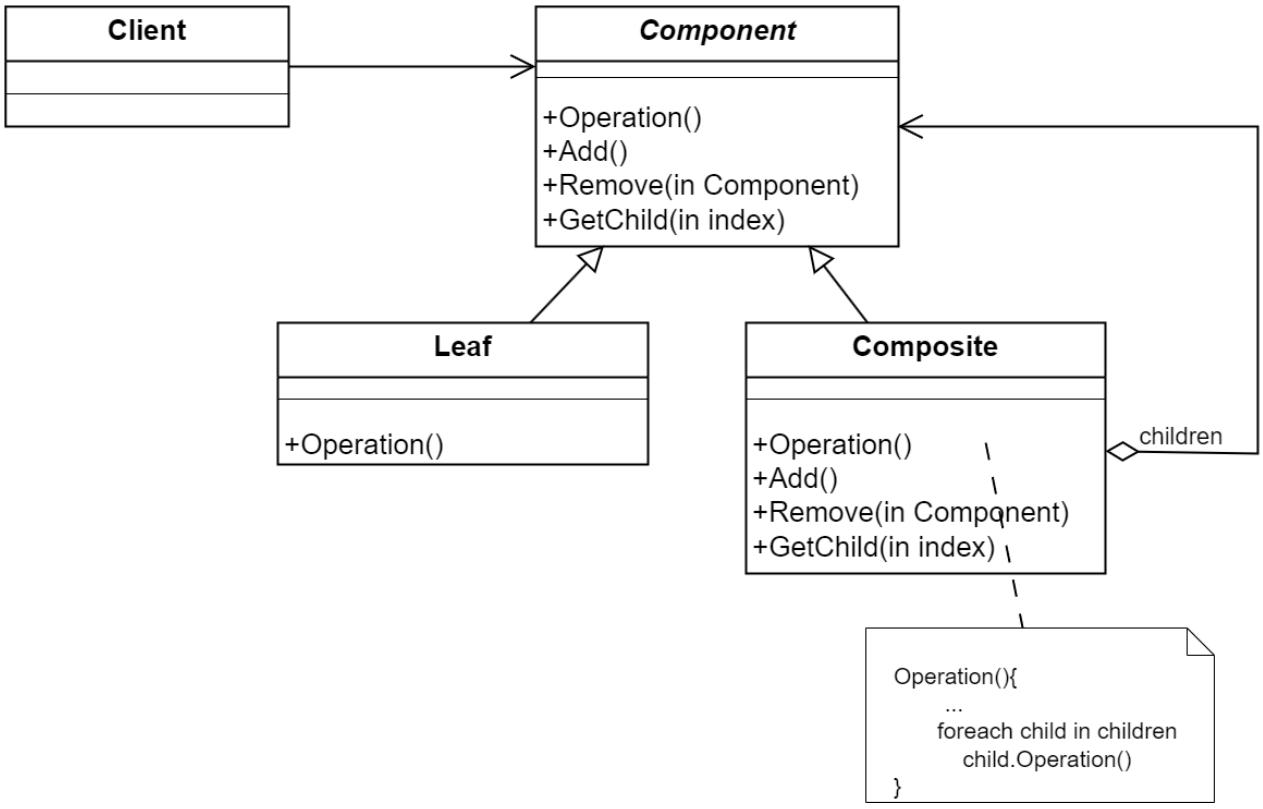

Рисунок 8.1. Структура патерна «Компонувальник»

Проблема: Ви розробляєте систему керування проєктами. Кожен проєкт складається із наборів функцій, кожна функція з userstory, а кожна userstory в свою чергу із задач по її реалізації. Ви реалізуєте функціонал відображення оціночної вартості робіт по кожній із функцій, а також відображення чи всі userstory були оцінені. Це потрібно бізнес-аналітики, щоб розуміти, що всі userstory розробниками були розглянуті і оцінені, а також обговорити необхідність реалізації того чи іншого функціоналу на основі попередньої оцінки.

Рішення: Крашим підходом в даній ситуації буде використання патерну Компоновщик. Класи що представляють функції, userstory, задачі будуть наслідуватися від одного інтерфейсу ITask, функції (Feature) та Userstory будуть складними об'єктами і міститимуть колекції об'єктів ITask, а задачі (Task) будуть представляли кінцеві об'єкти без дочірніх елементів.

Для розрахунку оціночної вартосі робіт, в ITask інтефейс додаємо метод GetEstimatedPoints(). В класах-компоновщиках методи GetEstimatedPoints() реалізовуємо як обхід всіх дочірніх елементів та сумування результатів відповідей GetEstimatedPoints().

Таким чином, візуальні форми будуть працювати з колекцією елементів ITask і їм не потрібно буде знати конкретні типи дочірніх класів з якими вони працюють. В результаті логіка візуальних форм виходить достатньо простою і вона не буде містити бізнес-логіки розрахунку загальної оцінки по проєктам, а просто викликає метод GetEstimatedPoints() не замислюючись містить цей об'єкт дочірні об'єкти чи ні.

Переваги та недоліки:

+ Спрощує представлення деревоподібної структури.

+ Додає гнучкості в роботі з складними об’єктами та рекурсивними операціями.

+ Дозволяє додавати та видаляти об’єкти в ієрархії без впливу на клієнтський код.

- Потрібні додаткові зусилля для початкового впровадження.

- Вимагає гарно спроєктованого загального інтерфейсу.

8.2.2. Шаблон «Flyweight»

Призначення: Шаблон використовується для зменшення кількості об'єктів в додатку шляхом поділу цих об'єктів між ділянками додатку. Flyweight являє собою поділюваний об'єкт.

Дуже важливою є концепція «внутрішнього» і «зовнішнього» станів [6]. Внутрішній стан відображає дані, характерні саме поділюваному об'єкту

(наприклад, код букви); зовнішній стан несе інформацію про його застосування в додатку (наприклад, рядок і стовпчик). Внутрішній стан зберігається в самому поділюваному об'єкті, зовнішній – в об'єктах додатку (контексту використання поділюваного об'єкта).

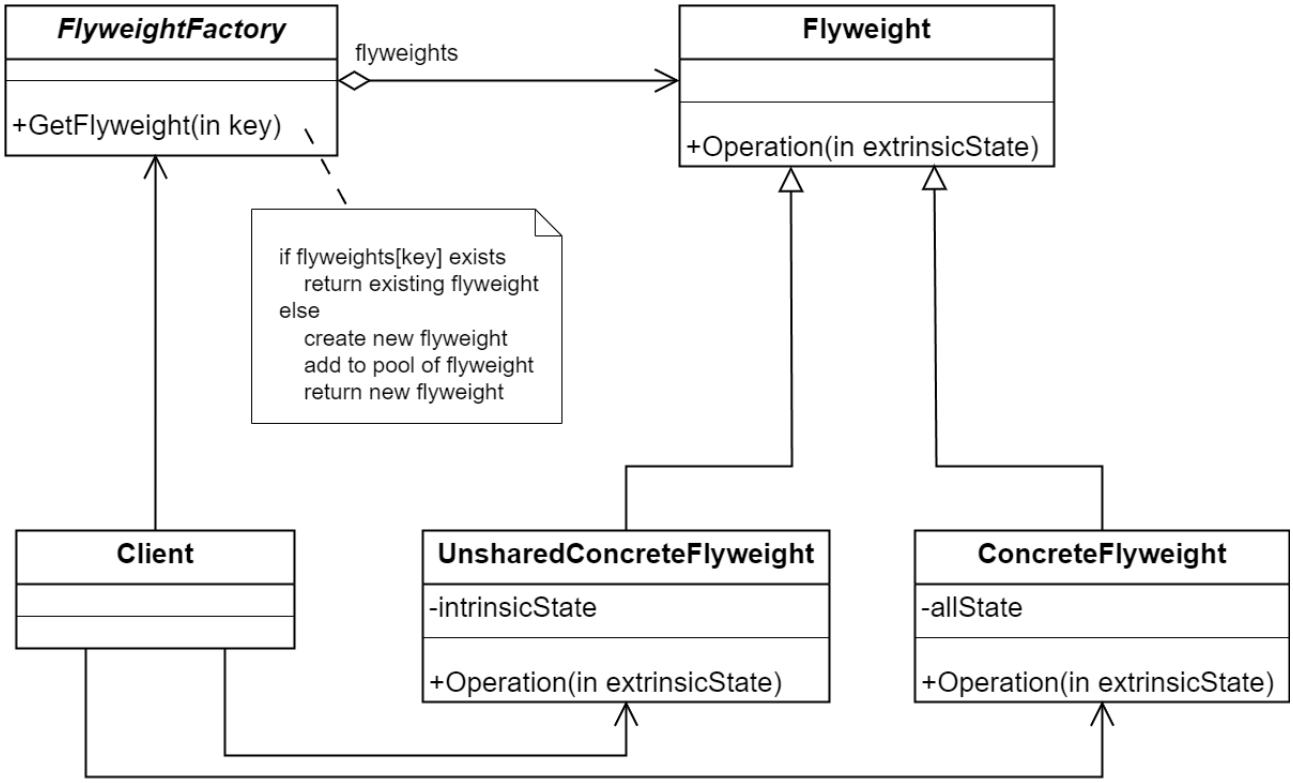

Рисунок 8.2. Структура патерну Flyweight (Легковаговик)

Хорошим прикладом є наступне зображення на рисунку 3.

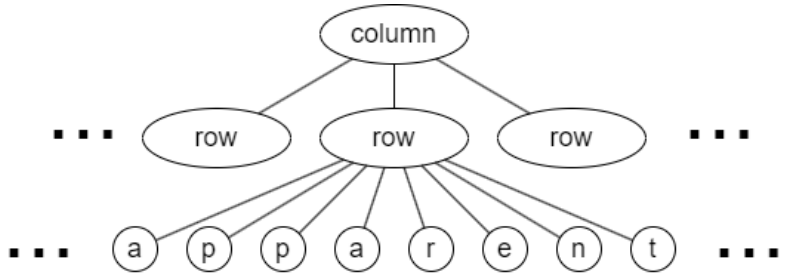

Рисунок 8.3. Приклад ситуації де варто використовувати Flyweight

Здається, ніби для кожної літери існує окремий об'єкт. Насправді фізично об'єкт всього один, існує лише безліч посилань на нього. (рисунку 8.4)

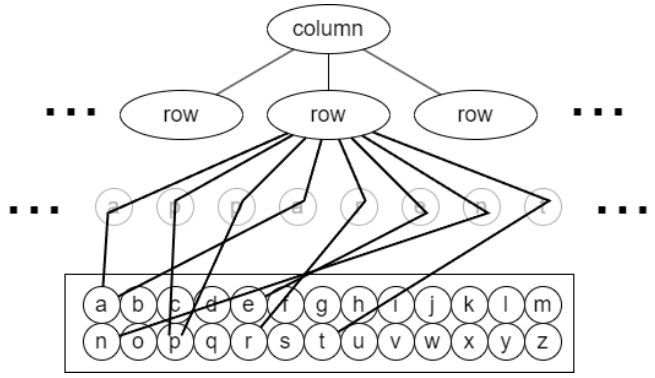

Рисунок 8.3. Використання патерну Flyweight

Даний шаблон дуже добре застосовувати у випадках, коли використовується безліч однакових об'єктів (наприклад, графічних примітивів).

Переваги та недоліки:

+ Заощаджує оперативну пам’ять.

- Витрачає процесорний час на пошук/обчислення контексту.

- Ускладнює код програми внаслідок введення безлічі додаткових класів.

8.2.3. Шаблон «Interpreter»

Призначення: Даний шаблон використовується для подання граматики і інтерпретатора для вибраної мови (наприклад, скриптової) [6]. Граматика мови представлена термінальними і нетермінальними символами, кожен з яких інтерпретується в контексті використання. Клієнт передає контекст і сформовану пропозицію в використовувану мову в термінах абстрактного синтаксичного дерева (деревоподібна структура, яка однозначно визначає ієрархію виклику підвиразів), кожен вираз інтерпретується окремо з використанням контексту. У разі наявності дочірніх виразів, батьківський вираз інтерпретує спочатку дочірні (рекурсивно), а потім обчислює результат власної операції.

Шаблон зручно використовувати в разі невеликої граматики (інакше розростеться кількість використовуваних класів) і відносно простого контексту (без взаємних залежностей і т.п.).

Даний шаблон визначає базовий каркас інтерпретатора, який за допомогою рекурсії повертає результат обчислення пропозиції на основі результатів окремих елементів.

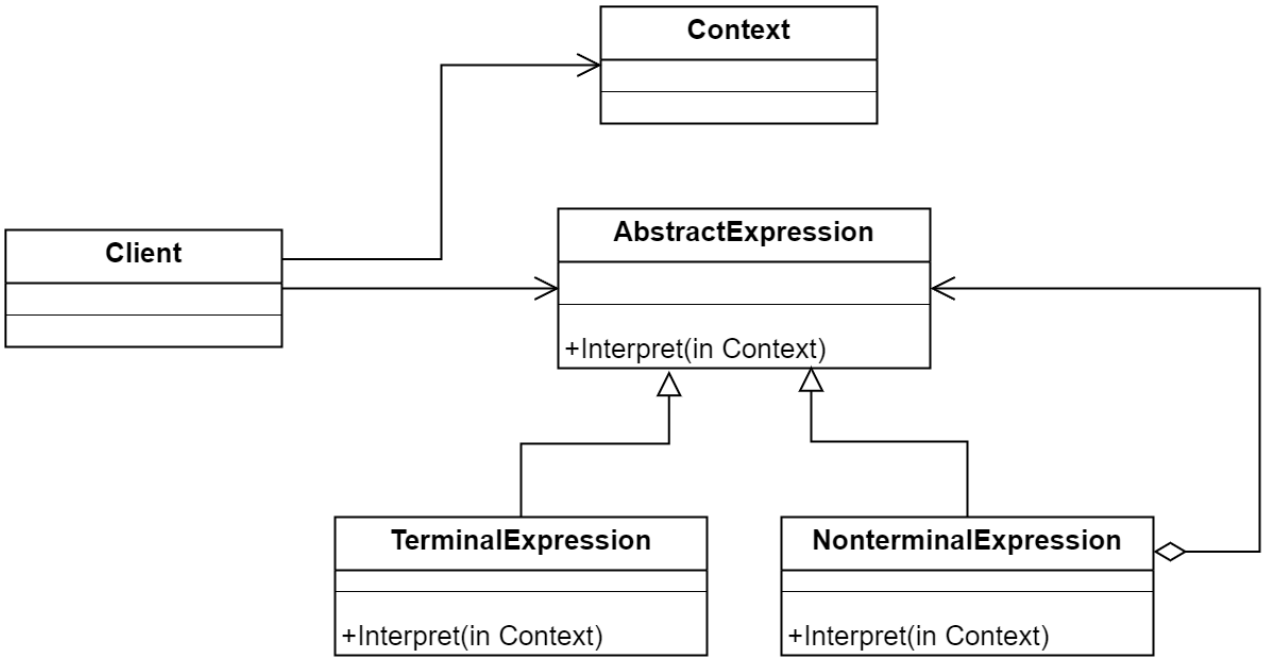

Рисунок 8.4. Структура патерна «Інтерпретатор»

При використанні даного шаблону дуже легко реалізовується і розширюється граматика, а також додаються нові способи інтерпретації виразів

Проблема: Стоїть задача пошуку рядків по зразку, як така, що часто зустрічається. Або ж загалом є якась задача, яка дуже часто змінюється.

Рішення: Може бути вирішена шляхом створення інтерпретатора, який визначає граматику мови. «Клієнт» будує речення у вигляді абстрактного синтаксичного дерева, у вузлах якого знаходяться об’єкти класів «НетермінальнийВираз» і «ТермінальнийВираз» (рекурсивне), потім «Клієнт» ініціалізує контекст і виражає операцію Розібрати(Контекст). На кожному вузлі типу «НетермінальнийВираз» визначається операція Розібрати для кожного підвиразу. Для класу «ТермінальнийВираз» операція Розібрати визначає базу рекурсії. «АбстрактнийВираз» визначає абстрактну операцію Розібрати,

загальну для всіх вузлів в абстрактному синтаксичному дереві. «Контекст» містить інформацію, глобальну по відношенню до інтерпретатора.

Переваги та недоліки:

+ Граматику стає легко розширювати та змінювати, реалізації класів, що описують вузли абстрактного синтаксичного дерева схожі (легко кодуються).

+ Можна легко змінювати спосіб обчислення виразів.

- Супроводження граматики с великою кількістю правил є проблематичним.

8.2.4. Шаблон «Visitor»

Призначення: Шаблон відвідувач дозволяє вказувати операції над елементами без зміни структури конкретних елементів [6]. Таким чином вкрай зручно додавати нові операції, проте дуже важко додавати нові елементи в ієрархію (необхідно додавати відповідні методи для обробки їх відвідувань в кожного відвідувача).

Даний шаблон дозволяє групувати однотипні операції, що застосовуються над різнотипними об'єктами.

Проблема: Ви розробляєте онлайн-корзину інтернет магазина. Товари які представлені в магазині є різних типів, наприклад, електроніка, міцні напої, домашня хімія.

Логіка роботи з товарами в корзині є різна, наприклад, розрахунок вартості, формування замовлення.

Ми можемо всі ці методи зробити в товарах, але тоді ми ускладнюємо товари і змішуємо логіку (розрахунок вартості) з даними (товаром та його кількістю).

Якщо ми в подальшому необхідно буде добавити ще логіку розрахунку вартості з врахуванням знижки, то потрібно буде добавити ще цю логіку до товарів. А якщо буде ще сезонна знижка, то ми знову будемо добавляти нову логіку до класів товарів.

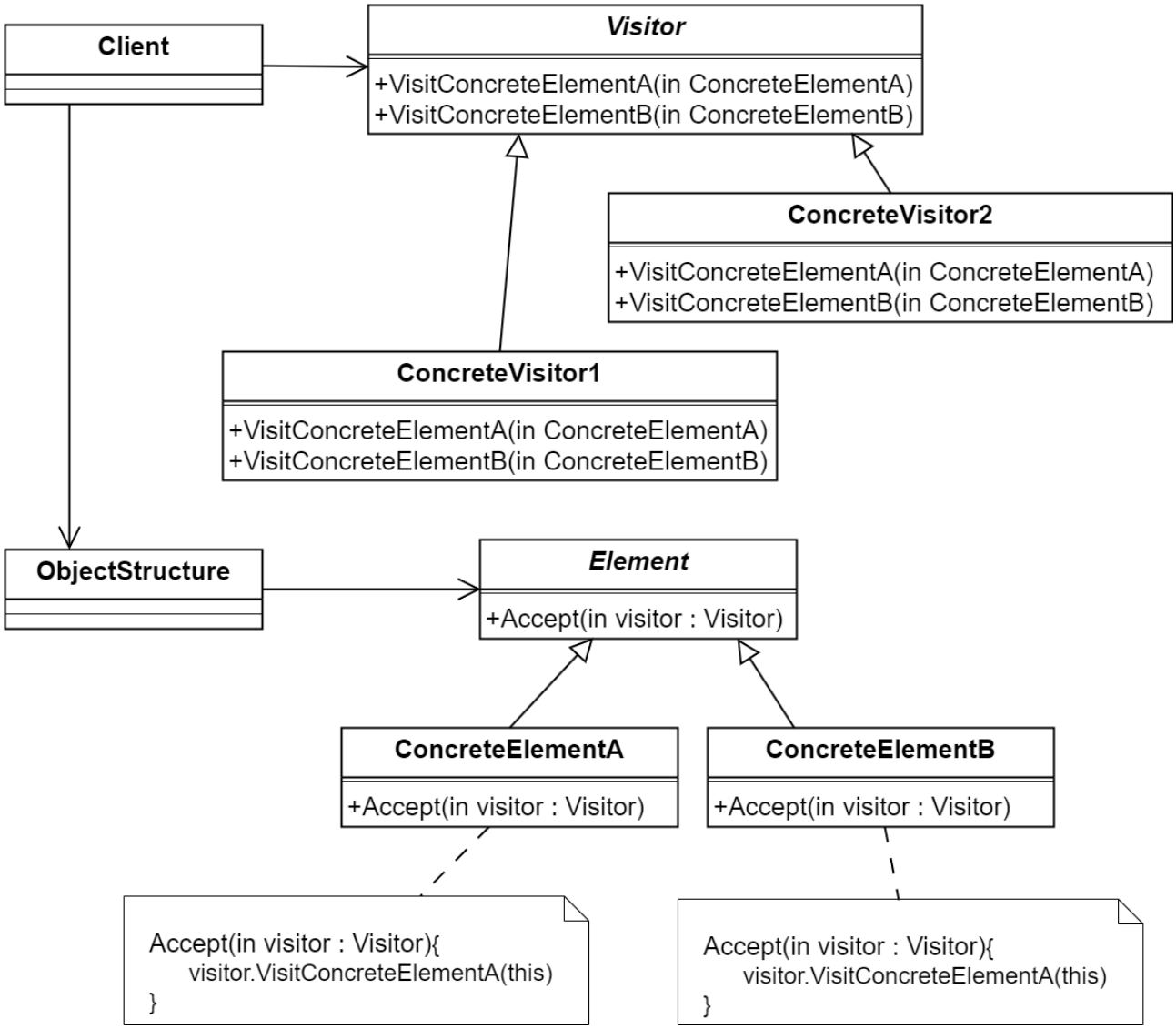

Рисунок 8.5. Структура патерна «Відвідувач»

Рішення: Основна ідея патерна «Відвідувач» це рознести логіку і дані в різні класи та ієрархії. Якщо так зробити для нашої онлайн-корзини, то в класі відвідувача ми маємо функції для обрахунку логіки для кожного типу, а об'єкти в корзині знають свій тип, отримують екземпляр відвідувача і в нього викликають метод відповідно до свого типу. В результаті ми робимо різні відвідувачі для розрахунку вартості: один для звичайних розрахунків, інший для розрахунку вартості зі знижкою, ще один для розрахунку сезонних знижок.

За рахунок того, що логіка відокремлена від наших товарів в корзині ми можемо реалізовувати за необхідності нові класи відвідувачі, а класи товарів та і корзини, в цілому, змінюватися не будуть.

Якщо розвивати далі, то можна зробити і клас відвідувача, який буде формувати замовлення з товарів в корзині в залежності від продавців і можливих варіантів доставки. Це дозволить формувати, наприклад, не одне замовлення, а два одна з самовивозом з магазину, а інше з доставкою Новою Поштою.

Приклад з життя: Прикладом може служити написання компілятора. Припустимо, існують різні об'єкти в синтаксисі мови програмування: виклики методів і умовні вирази. Компліятор перед генерацією коду повинен обійти всі вирази (і виклики методів, і умовні вирази) і перевірити безпеку типів, після чого згенерувати відповідний код. Відповідно буде два відвідувачі – для перевірки безпеки типів і для генерації коду. У кожного з них буде по 2 методи – для викликів методів і для умовних операцій. Таким чином при необхідності додавання нових кроків компіляції досить буде визначити нового «відвідувача» і викликати його у відповідний час.

8.3. Питання до лабораторної роботи

1. Яке призначення шаблону «Композит»?
2. Нарисуйте структуру шаблону «Композит».
3. Які класи входять в шаблон «Композит», та яка між ними взаємодія?
4. Яке призначення шаблону «Легковаговик»?
5. Нарисуйте структуру шаблону «Легковаговик».
6. Які класи входять в шаблон «Легковаговик», та яка між ними взаємодія?
7. Яке призначення шаблону «Інтерпретатор»?
8. Яке призначення шаблону «Відвідувач»?
9. Нарисуйте структуру шаблону «Відвідувач».
10. Які класи входять в шаблон «Відвідувач», та яка між ними взаємодія?

9. ЛАБОРАТОРНА РОБОТА №9

Тема: Взаємодія компонентів системи.

Мета: Вивчити види взаємодії додатків (Client-Server, Peer-to-Peer, Service-oriented Architecture), та реалізувати в проєктованій системі одну із архітектур.

9.1. Завдання

• Ознайомитись з короткими теоретичними відомостями.

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

• Реалізувати функціонал для роботи в розподіленому оточенні відповідно до обраної теми.

• Реалізувати взаємодію розподілених частин:

– *Для клієнт-серверних варіантів*: реалізація клієнтської і серверної частини додатків, а також загальної частини (middleware); зв'язок клієнтської і серверної частин за допомогою WCF, TcpClient, .NET-Remoting на розсуд виконавця.

– *Для однорангових мереж*: реалізація взаємодії клієнтських додатків за допомогою WCF Peer to peer channel.

– *Для SOA додатків*: реалізація сервісу, що надає послуги клієнтським застосуванням; викладання сервісу в хмару або підняття у вигляді Web Service на локальній машині; використання токенів для передачі даних про автентифікації, двостороннє шифрування.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє спроєктовану архітектуру. Навести фрагменти програмного коду, які є суттєвими для відображення реалізованої архітектури.

9.2. Теоретичні відомості

9.2.1. Клієнт-серверна архітектура

Клієнт-серверні додатки являють собою найпростіший варіант розподілених додатків, де виділяється два види додатків: клієнти (представляють додаток користувачеві) і сервери (використовується для зберігання і обробки даних). Розрізняють тонкі клієнти і товсті клієнти.

Тонкий клієнт – клієнт, який повністю всі операції (або більшість, пов'язаних з логікою роботи програми) передає для обробки на сервер, а сам зберігає лише візуальне уявлення одержуваних від сервера відповідей. Грубо кажучи, тонкий клієнт – набір форм відображення і канал зв'язку з сервером. Прикладом тонкого клієнта є класичні Web-застосунки.

У такому варіанті використання майже все навантаження лягає на сервер або групу серверів.

Перевагою таких моделей є простота розгортання, тому що оновлювати потрібно лише сервери і в результаті клієнти з наступними запитами автоматично будуть працювати з оновленою системою.

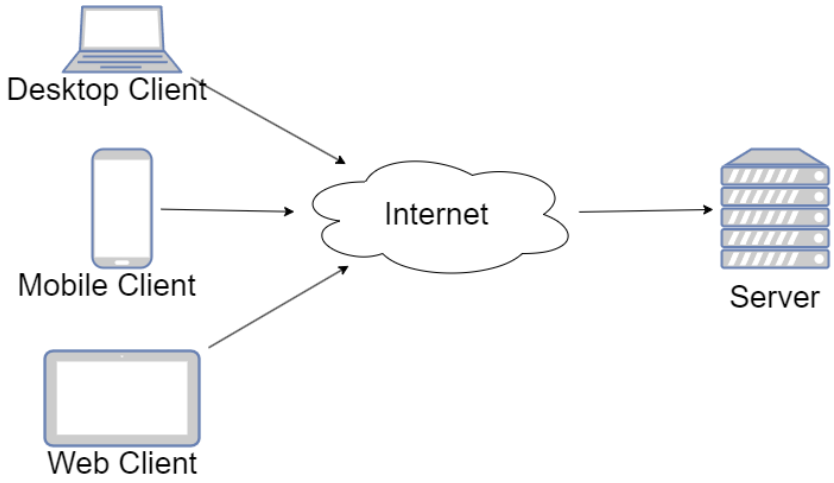

Рисунок 9.1. Клієнт-серверна архітектура

Товстий клієнт – антипод тонкого клієнта, більшість логіки обробки даних містить на стороні клієнта. Це сильно розвантажує сервер. Сервер в таких

випадках зазвичай працює лише як точка доступу до деякого іншого ресурсу (наприклад, бази даних) або сполучна ланка з іншими клієнтськими комп'ютерами. Перевагою такого підходу є менші вимоги до серверної частини. Також перевагою, при певному підході до реалізації, є можливість працювати клієнтам без тимчасового доступу до серверу. Прикладом товстого клієнта можна назвати мобільні застосунки, або десктоп застосунки. Наприклад, Evernote, Viber, MS Outlook, комп’ютерні антивіруси, ігри, що потребують інсталяції (The Sims, GTA, …) та інші.

Проміжним варіантом можна назвати SPA (Single Page Application) – це товсті Web-клієнти, які при старті кожен раз завантажуються з сервера, а надалі працюють з сервером через web-API. З одного боку більшу частину логіки вони відпрацьовують на клієнтський стороні, за рахунок чого зменшується серверне навантаження. Також оновлення простіше ніж для товстих клієнтів. Але такі застосунки не працюють, якщо сервер не доступний.

Клієнт-серверна взаємодія, як правило, організовується за допомогою 3-х рівневої структури: клієнтська частина, загальна частина, серверна частина.

Оскільки велика частина даних загальна (класи, використовувані системою), їх прийнято виносити в загальну частину (middleware) системи.

Клієнтська частина містить візуальне відображення і логіку обробки дії користувача; код для встановлення сеансу зв'язку з сервером і виконання відповідних викликів.

Серверна частина містить основну логіку роботи програми (бізнес-логіку) або ту її частину, яка відповідає зберіганню або обміну даними між клієнтом і сервером або клієнтами.

9.2.2. Peer-to-Peer архітектура

Peer-to-Peer (P2P) архітектура – це модель мережевої взаємодії, в якій кожен вузол (комп'ютер або пристрій) є одночасно клієнтом і сервером. У цій архітектурі всі вузли мають рівні права та можливості для обміну даними, ресурсами або виконання завдань. На відміну від клієнт-серверної моделі, де є

чітке розділення на клієнти й сервери, P2P-мережа дозволяє учасникам взаємодіяти безпосередньо, без необхідності в централізованому сервері.

Основними принципами P2P-архітектури є:

- Децентралізація – відсутність центрального сервера, що зменшує залежність від одного вузла, підвищуючи стійкість мережі до збоїв і атак.
- Рівноправність вузлів – кожен вузол може виконувати одночасно функції клієнта (отримувати ресурси) і сервера (надавати ресурси).
- Розподіл ресурсів – вузли надають доступ до своїх власних ресурсів, таких як обчислювальна потужність, дисковий простір або файли.

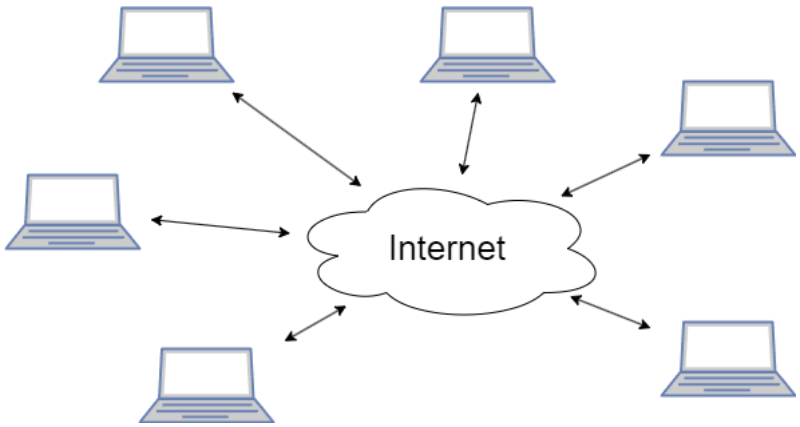

Рисунок 2. Peer-to-Peer архітектура

Основними сферами де peer-to-peer архітектура знайшла широке застосування є файлообмінники (BitTorrent), криптовалюти та інші блокчейн-технології, інтернет телефонія та відеоконференції (Skype, Zoom), розподілені обчислення (SETI@home, BOINC).

До основних проблемних зон можна віднести безпеку, синхронізацію даних та пошук ресурсів. Через централізацію складно контролювати дані, які передаються. Ефективність пошуку даних знижується зі збільшенням кількості вузлів у мережі і для підвищеня ефективності пошуку потрібно застосовувати спеціальні алгоритми.

9.2.3. Сервіс-орієнтована архітектура

Сервіс-орієнтована архітектура (SOA, англ. service-oriented architecture) – модульний підхід до розробки програмного забезпечення, заснований на використанні розподілених, слабо пов’язаних (англ. Loose coupling) сервісів або служб, оснащених стандартизованими інтерфейсами для взаємодії за стандартизованими протоколами [11].

Історично сервіс-орієнтована архітектура появилася як альтернатива монолітній архітектурі, в які вся система розроблялася та розгорталася як одне ціле.

Програмні комплекси, розроблені відповідно до сервіс-орієнтованою архітектурою, зазвичай реалізуються як набір веб-служб (або веб-сервісів), які, як правило, взаємодіють по HTTP з використанням SOAP або REST. Ці служби надають певні бізнес-функції, наприклад, отримання інформації про наявність матеріалів на складі.

Сервіси взаємодіють між собою тільки за рахунок обміну повідомленнями, без створення спеціальних інтеграцій для доступу до однієї інформації, наприклад, до однієї бази даних.

Сервіси також можуть бути реалізовані як обгортки навколо застарілої системи. Це робиться для зменшення вартості переробки системи, а також спрощення інтеграції існуючих монолітних систем в нову архітектуру.

Згідно SOA сервіси реєструються на спеціальних сервісах і будь-яка команда розробників, якій потрібен доступ може знайти їх та використовувати.

Часто реалізація SOA покладається на використання централізованого програмного компонента для обміну даними – шину даних (Enterprise Service Bus).

Мікросервісна архітектура є подальшим розвитком сервіс-орієнтованої архітектури з використанням нових напрацювань у інформаційних технологіях.

9.2.4. Мікро-сервісна архітектура.

Сама назва дає зрозуміти, що мікро-сервісна архітектура є підходом до створення серверного додатку як набору малих служб [11]. Це означає, що архітектура мікро-сервісів головним чином орієнтована на серверну частину, не дивлячись на те, що цей підхід так само використовується для зовнішнього інтерфейсу, де кожна служба виконується в своєму процесі і взаємодіє з іншими службами за такими протоколами, як HTTP/HTTPS, WebSockets чи AMQP. Кожен мікросервіс реалізує специфічні можливості в предметній області і свою бізнес-логіку в рамках конкретного обмеженого контексту, повинна розроблятись автономно і розвертатись незалежно.

Визначення мікросервісів із книги Іраклі Надарейшвілі, Ронні Мітра, Метта Макларті та Майка Амундсена (О'Рейлі) «Архітектура мікросервісів»:

«Мікросервіс – це компонент із чітко визначеними межами, який можна розгортати незалежно, і підтримує взаємодію за допомогою зв'язку на основі повідомлень. Архітектура мікросервісів – це стиль розробки високоавтоматизованих систем програмного забезпечення, що легко розвивати та яке складається з мікросервісів, орієнтованих на певні можливості».

Мікросервіси забезпечують чудові можливості супроводження в величезних комплексних системах з високою масштабуємістю за рахунок створення додатків, заснованих на множині незалежно розгортуючих служб з автономними життєвими циклами.

9.3. Питання до лабораторної роботи

1. Що таке клієнт-серверна архітектура?
2. Розкажіть про сервіс-орієнтовану архітектуру.
3. Якими принципами керується SOA?
4. Як між собою взаємодіють сервіси в SOA?
5. Як розробники взнають про існуючі сервіси і як робити до них запити?
6. У чому полягають переваги та недоліки клієнт-серверної моделі?

7. У чому полягають переваги та недоліки однорангової моделі взаємодії?

8. Що таке мікро-сервісна архітектура?

9. Які протоколи використовуються для обміну даними в мікросервісній архітектурі?

10. Чи можна назвати підхід сервіс-орієнтованою архітектурою, коли ми в проєкті між шаром веб-контролерів та шаром доступу до даних реалізуємо шар бізнес-логіки у вигляді сервісів?

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Prem Kumar Ponuthorai, Jon Loeliger. Version Control with Git, 3rd Edition – O'Reilly Media, 2022 – 546 p.
2. Pro Git, 2nd Edition. [Електронний ресурс] – Режим доступу: https://git-scm.com/book/en/v2
3. UML 2.5.1 Офіційна нотація [Електронний ресурс]. – Режим доступу: https://www.omg.org/spec/UML/About-UML/
4. Hugh Darwen, An Introduction to Relational Database Theory – ISBN 978-87-7681-500-4, 2010. – 231 p.
5. Ерік Фрімен, Елізабет Робсон. Head First. Патерни проєктування – Фабула, 2020 – 672с
6. Erich Gamma et al. Design Patterns: Elements of reusable object-oriented software – Addison-Wesley Professional, 1994 – 416p.
7. Martin Fowler. UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd Edition – Print2print, 2016. – 208
8. Refactoring guru. Шаблони проєктування [Електронний ресурс]. – Режим доступу: https://refactoring.guru/uk/design-patterns
9. Design Patterns in Java Tutorial [Електронний ресурс]. – Режим доступу: https://www.tutorialspoint.com/design_pattern
10. VanHakobyan. DesignPatterns. Приклади на C# [Електронний ресурс]. – Режим доступу: https://github.com/VanHakobyan/DesignPatterns
11. Mark Richard, Neal Ford. Fundamentals of Software Architecture – O'Reilly Media, 2020 – 422 p.

ПЕРЕЛІК ТЕМ ДЛЯ ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ

Поруч з темою перелічено які саме шаблони проєктування треба використати на кожну з лабораторних робіт відповідно

1. **Музичний програвач** (iterator, command, memento, facade, visitor, client-server)

Музичний програвач становить собою програму для програвання музичних файлів або відтворення потокової музики з можливістю створення, запам'ятовування і редагування списків програвання, перемішування/повторення (shuffle/repeat), розпізнавання різних аудіо-форматів, еквалайзер.

2. **HTTP-сервер** (state, builder, factory method, mediator, composite, p2p)

Сервер повинен мати можливість розпізнавати вхідні запити і формувати коректні відповіді (згідно протоколу HTTP), надавати сторінки chtml (html сторінки з додаванням найпростіших C# конструкцій на розсуд студента), вести статистику вхідних запитів, обробку запитів у багатопотоковому/подієвому режимах.

3. **Текстовий редактор** (strategy, command, observer, template method, flyweight, SOA)

Текстовий редактор повинен вміти розпізнавати текстові файли в будь-якій кодуванні, мати розширені функції редагування: макроси, сніппети, підказки, закладки, перехід на рядок / сторінку, підсвічування синтаксису (для однієї мови програмування або розмітки на розсуд студента).

4. **Графічний редактор** (proxy, prototype, decorator, bridge, flyweight, SOA)

Графічний редактор повинен вміти створювати / редагувати растрові (або векторні на розсуд студента) зображення в 2-3 основних популярних

форматах (bmp, png, jpg), мати панель інструментів для створення графічних примітивів, вибору кольорів, нанесення тексту, додавання найпростіших візуальних ефектів (ч/б растр, інфрачервоний растр, 2-3 на вибір учня), роботи з шарами.

5. **Аудіо редактор** (singleton, adapter, observer, mediator, composite, client-server)

Аудіо редактор повинен володіти наступним функціоналом: представлення аудіо даних будь-якого формату в WAVE-формі, вибір і подальші операції копіювання / вставки / вирізання / деформації по сегменту аудіозапису, можливість роботи з декількома звуковими доріжками, кодування в найбільш поширених форматах (ogg, flac, mp3).

6. **Web-browser** (proxy, chain of responsibility, factory method, template method, visitor, p2p)

Веб-браузер повинен мати можливість зробити наступне: мати адресний рядок для введення адреси сайту, переміщатися і відображати структуру html документа, переглядати підключений javascript та css файли, перегляд всіх підключених ресурсів (зображень), коректна обробка відповідей з сервера (коди відповідей HTTP) – переходи при перенаправленнях, відображення сторінок 404 і 502/503.

7. **Редактор зображень** (state, prototype, memento, facade, composite, client-server)

Редактор зображень має такі функціональні можливості: відкриття/збереження зображень у найпопулярніших форматах (5 на вибір студента), застосування ефектів, наприклад поворот, розтягування, стиснення, кадрування зображення, можливість створення колажів шляхом «нашарування» зображень.

8. **Powershell terminal** (strategy, command, abstract factory, bridge, interpreter, client-server)

Термінал для powershell повинен нагадувати типовий термінал з можливістю налаштування кольорів синтаксичних конструкцій, розміру вікна, фону вікна, а також виконання команд powershell і виконуваних файлів, а також працювати в декількох вікнах терміналу (у вкладках або одночасно шляхом розділення вікна).

9. **Sql IDE** (singleton, command, observer, bridge, visitor, SOA)

Інтегроване середовище розробки скриптів SQL повинна вміти працювати з більшістю сучасних баз даних (Microsoft SQL server, Oracle, IBM DB2, MySQL, PostgreSQL), виділяти синтаксичні одиниці конструкцій, виконувати і показувати результат їх виконання, отримувати схеми даних з джерела (список таблиць, стовпців, ключів і т.д.), порівнювати схеми даних з різних джерел.

10. **VCS all-in-one** (iterator, adapter, factory method, facade, visitor, p2p)

Клієнт для всіх систем контролю версій повинен підтримувати основні команди і дії (commit, update, push, pull, fetch, list, log, patch, branch, merge, tag) для 3-х основних систем управління версіями (svn, git, mercurial), а також мати можливість вести реєстр репозиторіїв (і їх типів) і відображати дерева фіксації графічно.

11. **Web crawler** (proxy, chain of responsibility, memento, template method, composite, p2p)

Веб-сканер повинен вміти розпізнавати структуру сторінок сайту, переходити за посиланнями, збирати необхідну інформацію про зазначений термін, видаляти не семантичні одиниці (рекламу, об'єкти javascript і т.д.), зберігати знайдені дані у вигляді структурованого набору html файлів вести статистику відвіданих сайтів і метадані.

12. **CI server** (state, command, decorator, mediator, visitor, soa)

–

13. **Office communicator** (strategy, adapter, abstract factory, bridge, composite, client-server)

Мережевий комунікатор для офісу повинен нагадувати функціонал програми Skype з можливостями голосового / відео / конференц-зв'язку, відправки текстових повідомлень і файлів (можливо, оффлайн), веденням організованого списку груп / контактів.

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.

15. **E-mail клієнт** (singleton, builder, decorator, template method, interpreter, client-server)

Поштовий клієнт повинен нагадувати функціонал поштових програм Mozilla Thunderbird, The Bat і т.д. Він повинен сприймати і коректно обробляти pop3/smtp/imap протоколи, мати функції автонастройки основних поштових провайдерів для України (gmail, ukr.net, i.ua), розділяти повідомлення на папки/категорії/важливість, зберігати чернетки незавершених повідомлень, прикріплювати і обробляти прикріплені файли.

16. **NAnt script builder** (proxy, chain of responsibility, decorator, mediator, flyweight, client-server)

Візуальна утиліта для створення скриптів NAnt збірки шляхом редагування текстового представлення xml-скрипта (з підсвічуванням синтаксису і

доповненням коду) або шляхом «перетягування» відповідних елементів; Також мати можливість автоматичного перетворення файлів .sln проєктів в скрипти збірки (шляхом аналізу проєктів і їх залежностей).

17. **System activity monitor** (iterator, command, abstract factory, bridge, visitor, SOA)

Монітор активності системи повинен зберігати і запам'ятовувати статистику використовуваних компонентів системи, включаючи навантаження на процесор, обсяг займаної оперативної пам'яті, натискання клавіш на клавіатурі, дії миші (переміщення, натискання), відкриття вікон і зміна вікон; будувати звіти про використання комп'ютера за різними критеріями (% часу перебування у веб-браузері, середнє навантаження на процесор по годинах, середній час роботи комп'ютера по днях і т.д.); правильно поводитися з «простоюванням» системи – відсутністю користувача.

18. **Shell (total commander)** (state, prototype, factory method, template method, interpreter, client-server)

Оболонка повинна вміти виконувати основні дії в системі – перегляд файлів папок в файлової системі, перемикання між дисками, копіювання, видалення, переміщення об'єктів, пошук.

19. **IRC client** (singleton, builder, abstract factory, template method, composite, client-server)

Клієнт для IRC-чатів з можливістю вказівки порту і адреси з'єднання, підтримка базових команд (підключення до чату, створення чату, установка імені, реєстрація, допомога і т.д.), отримання метаданих про канал.

20. **Mind-mapping software** (strategy, prototype, abstract factory, bridge, composite, SOA)

Візуальний додаток для складання "карт пам'яті" з можливістю роботи з декількома картами (у вкладках), автоматичного промальовування ліній, додавання вкладених файлів, картинок, відеофайлів (попередній перегляд); можливість додавання значків категорій / терміновості, обведення областей карти (поділ пунктирною лінією).

21. **Online radio station** (iterator, adapter, factory method, facade, visitor, client-server)

Додаток повинен служити сервером для радіостанції з можливістю мовлення на радіостанцію (64, 92, 128, 196, 224 kb/s) в потоковому режимі; вести облік підключених користувачів і статистику відвідувань і прослуховувань; налаштувати папки з піснями і можливість вести списки програвання або playlists (не відтворювати всі пісні).

22. **FTP-server** (state, builder, memento, template method, visitor, client-server)

FTP-сервер повинен вміти коректно обробляти і відправляти відповіді по протоколу FTP, з можливістю створення користувачів (з паролями) і доступних їм папок, розподілу прав за стандартною схемою (rwe), ведення статистики з'єднань, обмеження максимальної кількості підключень і максимальної швидкості поширення глобально і окремо для кожного облікового запису.

23. **Project Management software** (proxy, chain of responsibility, abstract factory, bridge, flyweight, client-server)

Програмне забезпечення для управління проєктами повинно мати наступні функції: супровід завдань/вимог/проєктів, списків команд, поточних завдань, планування за методологіями agile/kanban/rup (включаючи дошку завдань, ітерації тощо), мати можливість прикріплювати вкладені файли до завдань та посилатися на конкретні версії програми, зберігати виконувані файли для кожної версії.

24. **Flexible automation tool** (strategy, command, abstract factory, facade, interpreter, SOA)

Інструмент автоматизації повинен забезпечувати найпростіші автоматичні дії для зручності користувача: завантаження нових фільмів / книг / файлів при випуску (наприклад, щоп'ятниці з'являються нові серії улюблених серіалів); встановити статуси в комунікаторах (skype – away при нульовій активності на тривалий час) і т.д. Автоматизація забезпечується шляхом введення правил (на зразок IFTTT.com сервісу), запису макросів (натискання клавіш, дії миші), планувальника завдань (о 5 ранку – початок роздачі торрент-файлів).

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.

26. **Download manager** (iterator, command, observer, template method, composite, p2p)

Інструмент для скачування файлів з інтернету по протоколах http або https з можливістю продовження завантаження в зупиненому місці, розподілу швидкостей активним завантаженням, ведення статистики завантажень, інтеграції в основні браузери (firefox, opera, internet explorer, chrome).

27. **Особиста бухгалтерія** (state, prototype, decorator, bridge, flyweight, SOA)

Програма повинна бути наочним засобом для ведення особистих фінансів: витрат і прибутку; з можливістю встановлення періодичних витрат /

прибутку (зарплата і орендна плата); введення сканованих чеків з відповідними статтями витрат; побудова статистики; експорт/імпорт в Excel, реляційні джерела даних; різні рахунки; ведення єдиного фонду на всі рахунки (всією сім'єю) – на особливі потреби (ремонт, автомобіль, відпустка); можливість введення вкладів / кредитів для контролю банківських рахунків (звірка нарахованих відсотків з необхідними і т.д.).

28. **JSON Tool (ENG)** (strategy, command, observer, template method, flyweight)

Display JSON schema with syntax highlight. Validate JSON schema and display errors. Create user friendly table\list box\other for read and update JSON schema properties metadata (description, example, data type, format, etc.). Auto save\restore when edit, maybe history. Can check JSON value by schema (Put schema and JSON = valid\invalid, display errors). Export schema as markdown table. JSON to "flat" view.

29. **Ваш варіант** (тему та шаблони проєктування узгодити із викладачем)