Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Аналіз та використання патернів проектування

Текстова частина до курсової роботи за спеціальністю

«Комп’ютерні науки» 122

Керівник курсової роботи

Яремко С. А.

_____

(підпис)

“____” _____ 2021 р.

Виконала студентка

Харченко М. В.

“____” _____ 2021 р.

Київ 2021

Індивідуальне завдання

на курсову роботу

Студентці Харченко Марині Вадимівні факультету інформатики 4 курсу

ТЕМА: Аналіз та використання патернів проектування

Вихідні дані:

Зміст ТЧ до курсової роботи:

Календарний план

Вступ

Розділ 1: Породжуючі патерни

Розділ 2: Структурні патерни

Розділ 3: Поведінкові патерни

Висновки

Список використаної літератури

Тема: Аналіз та використання патернів проектування

Календарний план виконання роботи:

№	Назва етапу	Термін виконання	Примітка
1.	Отримання теми курсової роботи	22.10.2020	
2.	Пошук тематичної літератури	5.11.2020	
3.	Ознайомлення з літературою	22.11.2020	
4.	Ознайомлення з теоретичним матеріалом	12.01.2021	
5.	Пошук прикладів застосування патернів у існуючих фреймворках	15.02.2021	
9.	Написання першого розділу теоретично і практичної частин	28.02.2021	
10.	Написання другого розділу теоретично і практичної частин	9.03.2021	
11.	Написання третього розділу теоретично і практичної частин	27.03.2021	
13.	Внесення змін до курсової роботи відповідно до зауважень наукового керівника	5.04.2021	
14.	Створення презентації	9.04.2021	
15.	Захист роботи	18.04.2021	

Студент Харченко М. В.

Керівник Яремко С. А.

“____” _____

Зміст

Анотація

Під час виконання даної курсової роботи було проаналізовано 23 типових патерна проектування, особливу увагу було приділено їх розділенню за типом призначення на: породжуючі, структурні та поведінкові. В результаті, було спроектовано 3 системи, кожна з яких поєднує декілька патернів одного типу.

Вступ

При розробці програмного забезпечення, патерни програмування мають два основні призначення. Перше з них - це забезпечення стандартів розробки. Патерни програмування забезпечують дотримання стандартної термінології в визначених сценаріях. Другим призначенням патернів є формулювання принципів для забезпечення найкращих практик при написанні коду. Вивчення та застосування патернів програмування при розробці, дозволяє підвисити надійність та якість розроблених застосунків.

Стандартні патерни проектування поділяються за призначенням на 3 типи: породжуючі, структурні та поведінкові. Перед використанням патерну важливо розуміти, яку проблему він повинен вирішувати за своїм типом призначення. Іноді доцільним є поєднання декількох патернів для досягнення певних цілей у роботі системи та дотримання певних загальноприйнятих стандартів в організації коду.

В даній роботі надається коротка характеристика кожному з 23 патернів проектування, більше уваги приділяється 6 патернам, а саме: Абстрактній фабриці, Одинаку, Адаптеру, Фасаду, Команді та Спостерігачу. Вибір цих патернів обумовлено частотою їх використання, опираючись на дані, зібрані у літературі [4]. Також, було створено 3 системи, що демонструють одночасне використання патернів проектування одного типу.

Актуальність теми:

На цей час майже відсутні навіть суто демонстраційні приклади систем, в яких кілька патернів одного типу працюють поряд або разом. Зазвичай, патерни одного типу прийнято вважати антагоністами.

Метою роботи є:

Аналіз стандартних патернів проектування та їх обґрунтоване використання при проектуванні систем, для вирішення певних конкретні задач.

Структура роботи:

Робота має три розділи:

1) В першому розділі надається коротка характеристика породжуючих патернів проектування та розглядається поєднання патернів Абстрактна фабрика й Одинак в одній системі.
2) В другому розділі надається коротка характеристика структурних патернів проектування та розглядається поєднання патернів Фасад й Адаптер в одній системі
3) В третьому розділі надається коротка характеристика поведінкових патернів проектування та розглядається поєднання патернів Фасад й Адаптер в одній системі

Розділ 1. Породжуючі патерни

1.1 Призначення породжуючих патернів

Породжуючі патерни проектування абстрагують процес створення екземплярів класів. Вони допомагають зменшити залежність та зв’язність між собою процесів створення, компонування та представлення об’єктів. Породжуючі патерни набувають більшої важливості по мірі розростання та ускладнення системи, коли композиція починає превалювати над наслідуванням. Як наслідок, жорстке програмування поведінки всієї системи стає недоцільним і перевага надається визначенню невеликого набору простих фундаментальних поведінок, які потім можуть бути скомпоновані у будь-яку кількість більш складних.

Патерни рівня класів використовують наслідування, для внесення варіативності та гнучкості при створенні екземплярів класів. Патерни рівня об’єктів делегують екземплярам класів процес створення інших екземплярів класів.

Породжуючі патерни виконують дві важливі функції. По-перше, вони інкапсулюють знання про які точно класи є відповідальними за створення екземплярів. По-друге, вони ховають знання про те, як екземпляри будуть створені та скомпоновані. Глобально, все що знає система про об'єкти, так це їх інтерфейси, які можуть бути визначені абстрактними класами.

Додатково, патерни проектування поділяються на два типи: патерни рівня класів та патерни рівня об’єктів. Патерни проектування рівня класів направлені на організацію відношень між класами. Для таких патернів характерне використання статичних зв'язків: наслідування та реалізації. Патерни проектування рівня об’єктів описують моделі взаємодії між об’єктами. Для таких патернів характерне використання динамічних зв'язків: асоціації, агрегації та композиції.

Отже, породжують патерни дають більше гнучкості у визначенні того, *що* буде створено, *ким* буде

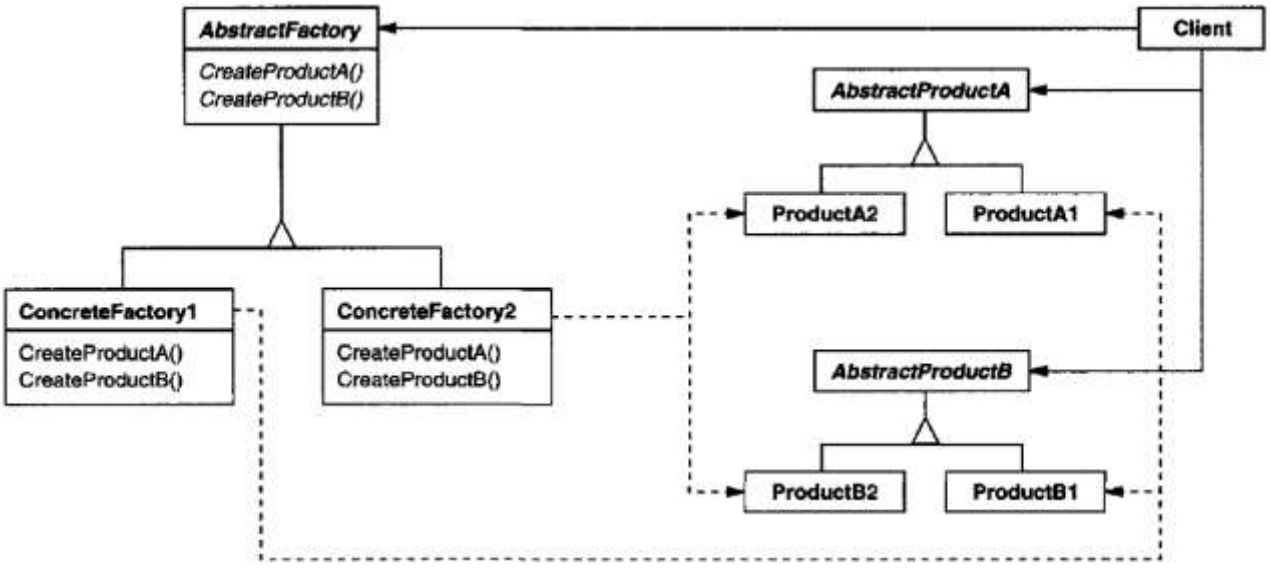
1.2 Характеристика існуючих породжуючих патернів

Існує 5 породжуючих патернів:

1. Абстрактна фабрика (abstract factory).

   Забезпечує наявність інтерфейсу через який будуть створені споріднені або залежні об’єкти-продукти, не надаючи інформацію про конкретні класи цих об’єктів. Абстрактна фабрика є патерном рівня об’єктів.

   Структура:



(Рисунок 1.1 – загальна структура патерну Абстрактна фабрика [1]).

Патерн використовується, коли:

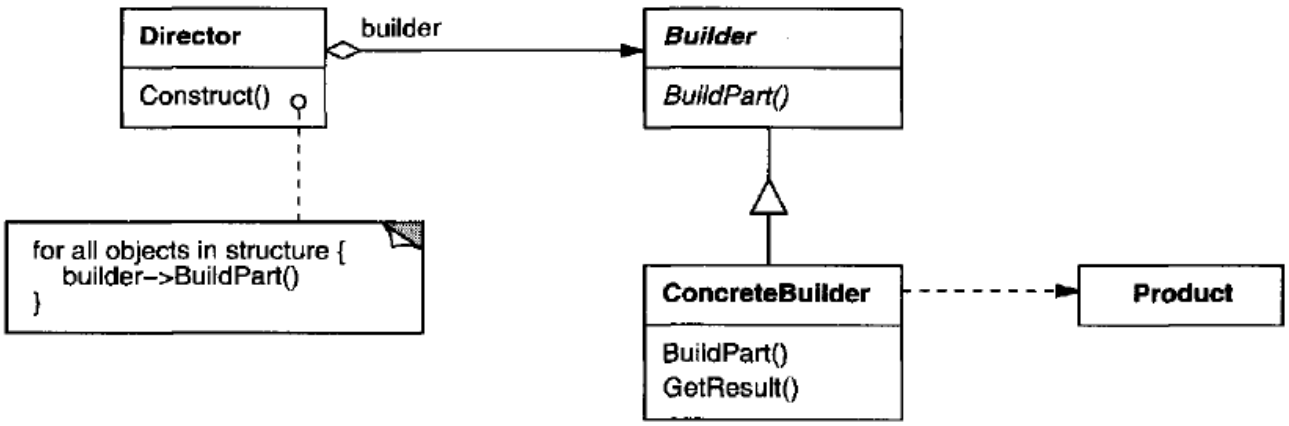
- в системі використовується одна чи декілька множин об’єктів-продуктів, згрупованих за походженням.

- множина, згрупованих за походженням, об’єктів-продуктів повинна використовуватись разом, й потрібно закріпити це правило у системі.
- потрібно створити бібліотеку класів, що міститиме об’єкти-продукти, згруповані за походженням, та потрібно надати лише їх інтерфейси без імплементації.

2. Будівельник (builder).

   Відокремлює логіку будування складних об’єктів від їх представлення. Таким чином один й той же процес будування може використовуватись для створення різних представлень цих об’єктів. Будівельник є патерном рівня об’єктів.

   Структура:



(Рисунок 1.2 – загальна структура патерну Будівельник [1])

Патерн використовується, коли:

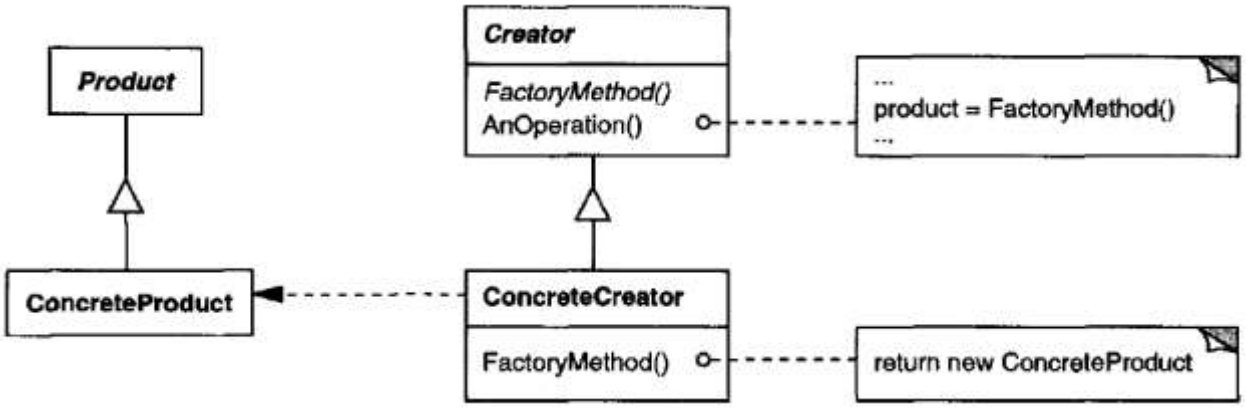
- загальний алгоритм породження складного об’єкту повинен бути незалежним від певних окремих його частин.

- процес створення об’єкту повинен дозволяти створювати різні представлення об’єктів.

3. Фабричний метод (factory method).

   Визначає абстрактний інтерфейс для створення об’єктів, проте саме класи-наслідники відповідають за визначення який саме тип та конфігурація об’єкта-продукту буде створюватись. Фабричний метод є патерном рівня класів.

   Структура:



(Рисунок 1.3 – загальна структура патерну Фабричний метод [1])

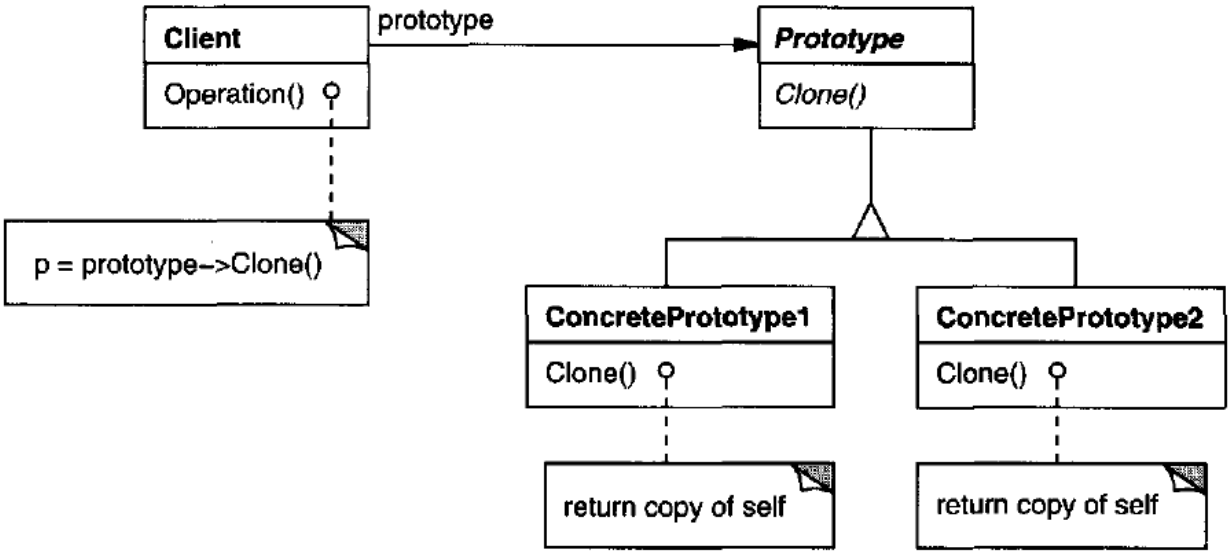
Патерн використовується, коли:

- класу заздалегідь не відомо об’єкти якого класу йому потрібно породити.
- відповідальність за визначення типу об’єкта-продукту повинна належати класам-наслідникам.
- класи повинні делегувати відповідальність за визначення типів об’єктів-продуктів декільком класам-помічникам. І потрібно локалізувати знання про те, якому класу-помічнику буду делегована відповідальність.

4. Прототип (prototype).

   Надає можливість створювати нові об’єкти, використовуючи екземпляр-прототип, прибираючи необхідність надавати значення всім полям та властивостям нового екземпляру. Прототип є патерном рівня об’єктів.

   Структура:



(Рисунок 1.4 – загальна структура патерну Прототип [1])

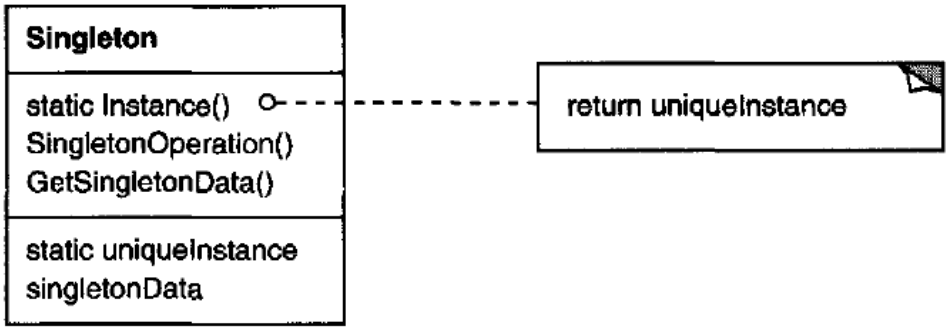
Патерн використовується, коли:

- заздалегідь невідомо екземпляр якого класу потрібно буде створити, наприклад, при динамічному завантаженні.
- потрібно уникнути створення паралельної ієрархії класів
- потрібно зменшити кількість класів, що відрізняються лише тим, як будуть ініціалізовані їх об’єкти.

5. Одинак (singleton).

   Забезпечує існування лише єдиного екземпляру класу та існування єдиного глобального доступу до цього екземпляру. Одинак є патерном рівня об’єктів.

   Структура:



(Рисунок 1.5 – загальна структура патерну Одинак [1])

Патерн використовується, коли:

- потрібен лише один екземпляр класу (у деяких випадках може використовуватись декілька екземплярів. Проте їх кількість завжди обмежується на етапі проектування системи.)
- потрібно локалізувати доступ до екземпляру класу в одній глобальній точці у системі.

1.3 Приклад компонування кількох породжуючих патернів в одну систему

Завдання: спроектувати бота, що самостійно зможе торгувати парами валют. Інформацію про те, які саме пари валют будуть використовуватись, бот буде отримувати з зовнішнього ресурсу. Система повинна буде працювати у вигляді окремого процесу, та створювати у межах себе окремі потоки. Кожен з створених

потоків займається торгівлею на біржі, роблячи транзакції, коли це є вигідним та зберігаючи кожну свою дію у базу даних.

Постановка задачі проектованої системи: від системи потребується гнучкість у створенні невизначеною заздалегідь кількості споріднених об’єктів-продуктів. Споріднені об’єкти-продукти будуть користуватись спільним API. Проте API буде різнитись для кожного типу валют. Додатково, система повинна впровадити єдиний екземпляр DbConnectionPool, який буде глобально доступним для кожного екземпляру продукту.

При проектуванні такої системи доцільно скористатися патернами Абстрактна фабрика та Одинак.

1.3.1 Патерн абстрактна фабрика (abstract factory)

Роль патерну Абстрактна фабрика у системі: побудова правильної ієрархії та надання уніфікованого інтерфейсу, що надає можливість створювати споріднені об’єкти, у заздалегідь невідомій кількості, проте з переважно відомим внутрішнім представленням.

Приклади використання патерну Абстрактна фабрика у реальних системах:

Абстрактна фабрика	
Фреймворк	Використання
JavaBean Framework	Забезпечує інтерфейс для створення груп об'єктів без зазначення їх конкретних класів. Наприклад є інтерфейси InitialContext, InitialContextFactory.

	InitialContextFactory, що мають методи отримання InitialContext.
.NET Framework	Використовується в ADO.NET, визначає набір абстрактних класів, таких як DbConnection, DbCommand, DbParameter. У той же час кожен з них має конкретний клас, такий як SqlConnection або OralceConnection. Класи XxxConnection успадковують і реалізують DbConnection.

(Таблиця 1.1 – використання патерну Абстрактна фабрика у реальних системах)

1.3.2 Патерн Одинак (singleton)

Роль патерну Одинак у системі: патерн слідкує за існуванням одного та лише одного database connection pool у системі. Єдиний екземпляр створюється одразу при запуску системи. Кожен з конкретних потоків може користуватись з'єднаннями з базою, що надає пул, та відпускати, коли потреби більше у ньому не має. Доступ до цього об'єкту повинен бути одним і бути глобально доступним.

Приклади використання патерну Одинак у реальних системах:

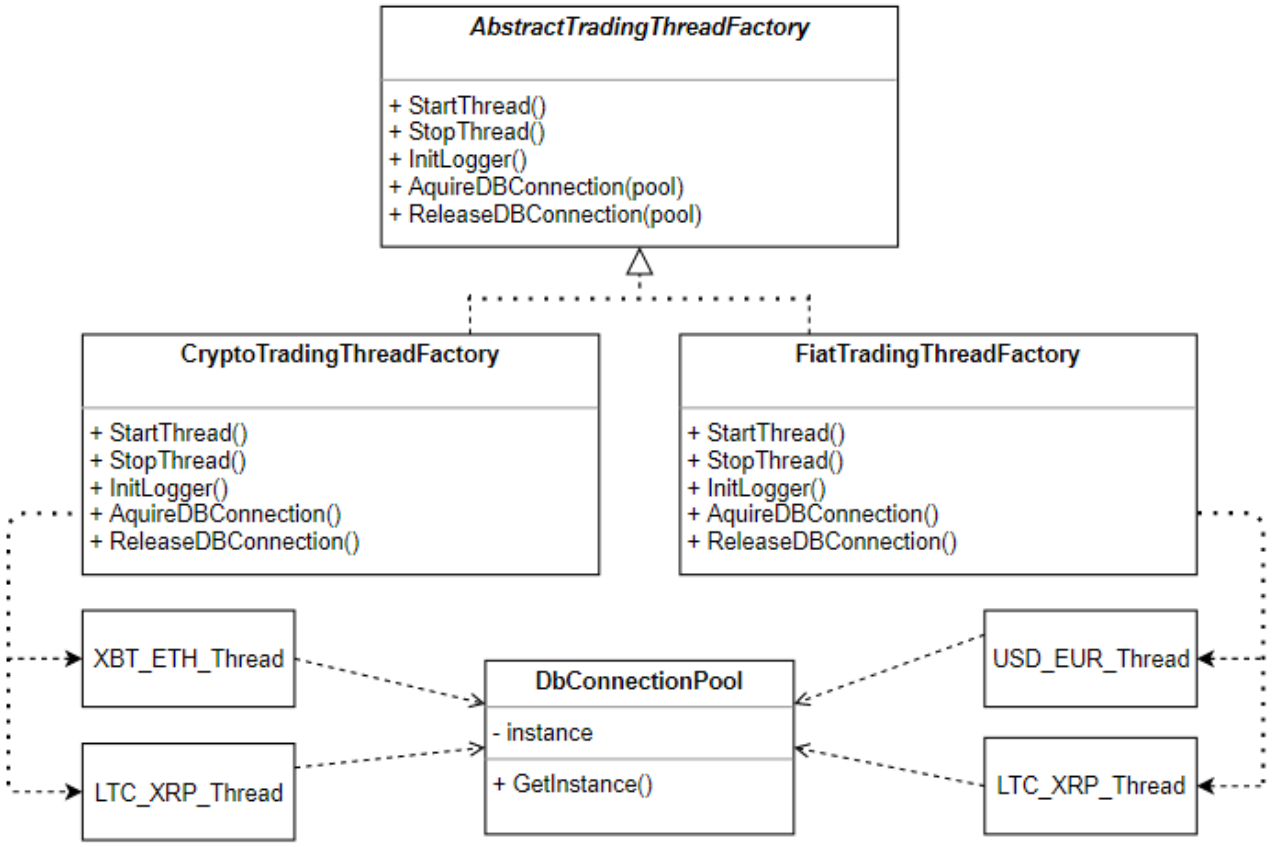
Одинак	
Фреймворк	Використання
JavaBean Framework	Переконується, що клас має лише один екземпляр, і надайте глобальний доступ до нього.

	Таких класів багато. Одним із прикладів є javax.naming.NamingManager
Spring Framework	За замовчуванням кожен написаний вами клас Spring це Одинак. Ви можете використовувати його в безлічі місць, і всі вони будуть встановлені автоматично шляхом введення, і всі 100 посилань будуть спрямовані на один і той же об’єкт (тобто синглтон).

(Таблиця 1.2 – використання патерну Одинак у реальних системах)

1.3.3 Компонування патернів у систему

При вирішенні сформульованої задачі за допомогою патернів Абстрактна фабрика та Одинак була спроектована така система:



(Рисунок 1.3.3 – спроектована система за сформульованим завданням)

1.4 Висновок

У даному розділі були проаналізовані породжуючі патерни проектування. Особлива увага була приділена патернам Абстрактна фабрика та Одинак. Була розглянута на конкретному прикладі доцільність поєднання цих двох патернів при вирішенні конкретної задачі. Ми з'ясували, що їх поєднання може зробити систему більш надійною, гнучкою та покращити якість розробленої системи. Отже при використанні одного породжуючого патерну треба пам'ятати про можливість поєднання його з іншим патерном цього типу.

Розділ 2. Структурні патерни

2.1 Призначення структурних патернів

Структурні патерни спрямовані на компоновку класів і об'єктів в більш складні структури.

Патерни рівня класів використовують успадкування для комбінування інтерфейсів і реалізацій у системі. Простим прикладом може бути множинне наслідування інтерфейсів або множинне наслідування класів. Множинне наслідування інтерфейсів дозволяє організовувати композиції інтерфейсів; множинне наслідування класів дозволяє створювати класи-наслідники, що об’єднують властивості всіх своїх батьківських класів. Патерни рівня об’єктів використовують композицію об’єктів для отримання нової функціональності. Таким чином, ми отримуємо додаткову гнучкість, яка пов’язана з можливістю змінити композицію об’єктів під час виконання, що є неможливим у випадку статичної композиції класів.

Структурні патерни особливо корисні у ситуаціях, коли ми хочемо змусити працювати в одній системі модулі, що були незалежно розроблені один від одного.

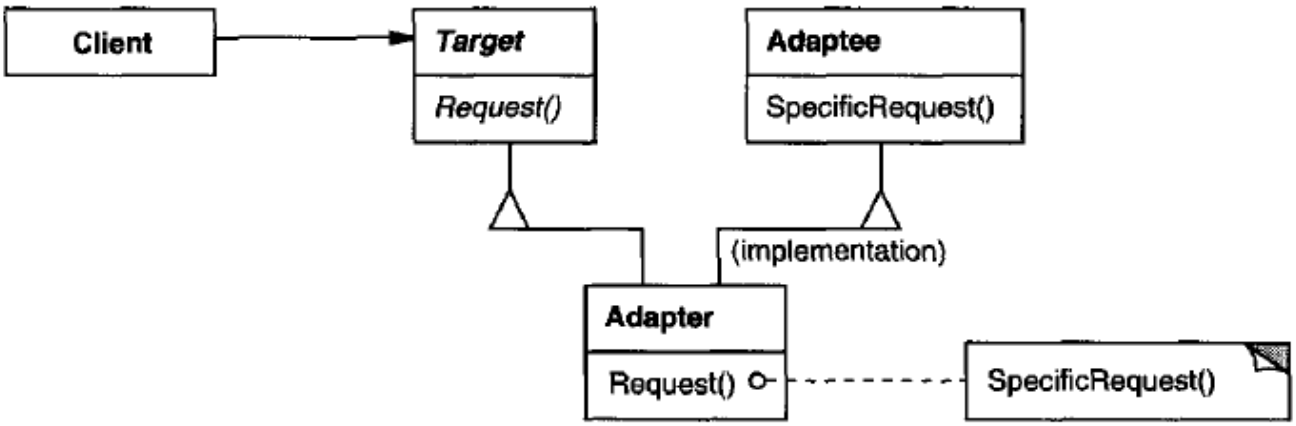
2.2. Характеристика існуючих структурних патернів

Існує 7 структурних патернів:

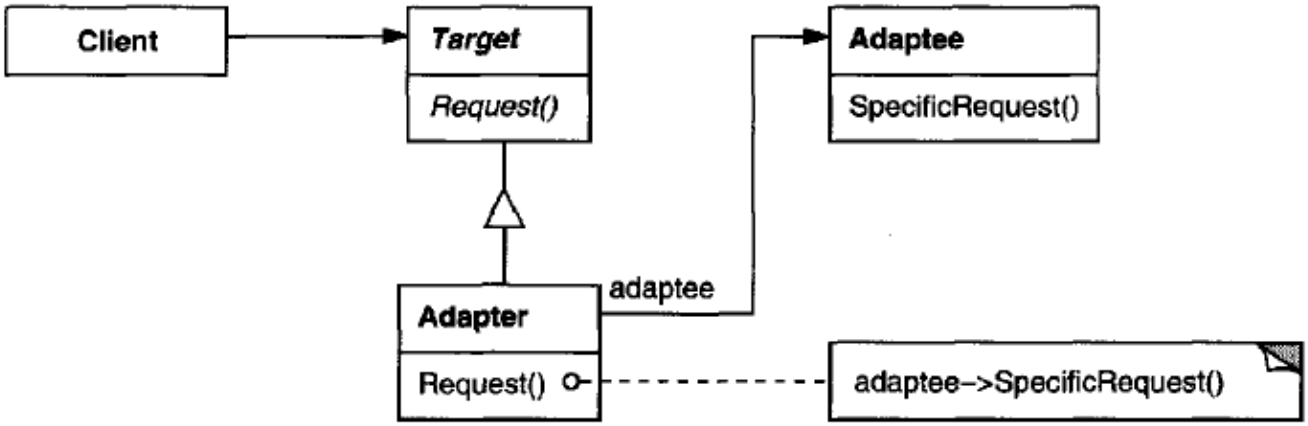
1. Адаптер (Adapter).
   Перетворює інтерфейс (як набір властивостей та методів) одного класу в інтерфейс другого класу, який буде очікуватись клієнтами. Цей патерн

забезпечує сумісну роботу класів з несумісними інтерфейсами. Адаптер є патерном рівня об’єктів та класів.

Структура:



(Рисунок 2.1.1 – загальна структура патерну Адаптер рівня класів [1])



(Рисунок 2.1.2 – загальна структура патерну Адаптер рівня об’єктів [1])

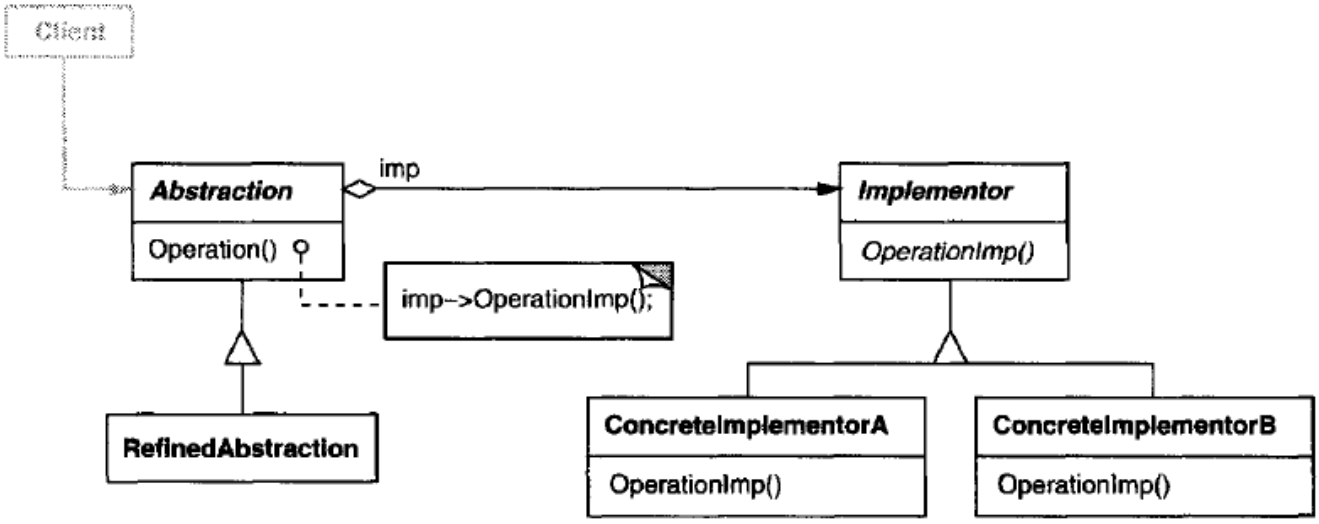
Патерн використовується, коли:

- потрібно використати вже існуючий у системі клас, проте його інтерфейс не відповідає очікуванням клієнта.
- потрібно створити клас, що буде повторно використовуватись з класами, що мають несумісні інтерфейси.

2. Міст (Bridge).

   Забезпечує незалежність рівня абстракції класу від елементів його реалізації, так, щоб абстракцію і реалізацію можна було змінювати незалежно одне від одного. Міст є патерном рівня об’єктів.

   Структура:



(Рисунок 2.2 – загальна структура патерну Міст [1])

Патерн використовується, коли:

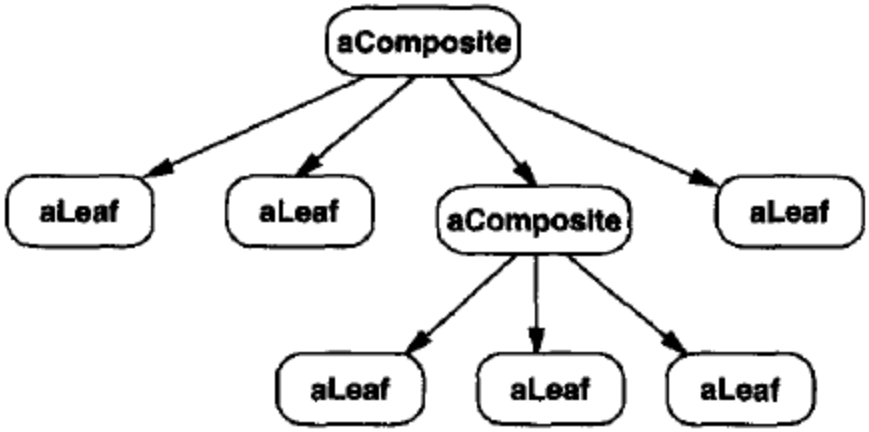
- потрібно уникнути постійної прив’язки між абстракцією та імплементацією. Наприклад, коли потрібно підмінити імплементацію під час виконання.

- одночасно абстракція та імплементація повинні бути доступними для розширення незалежно один від одного.

3. Компонувальник (Composite).

   Дозволяє будувати ієрархії із об’єктів у вигляді дерев. Надаючи можливість клієнтському коду однаково трактувати окремі об’єкти (листя) й складені об’єкти (гілки). Компонувальник є патерном рівня об’єктів.

   Структура:



(Рисунок 2.3 – загальна структура патерну Компонувальник [1])

Патерн використовується, коли:

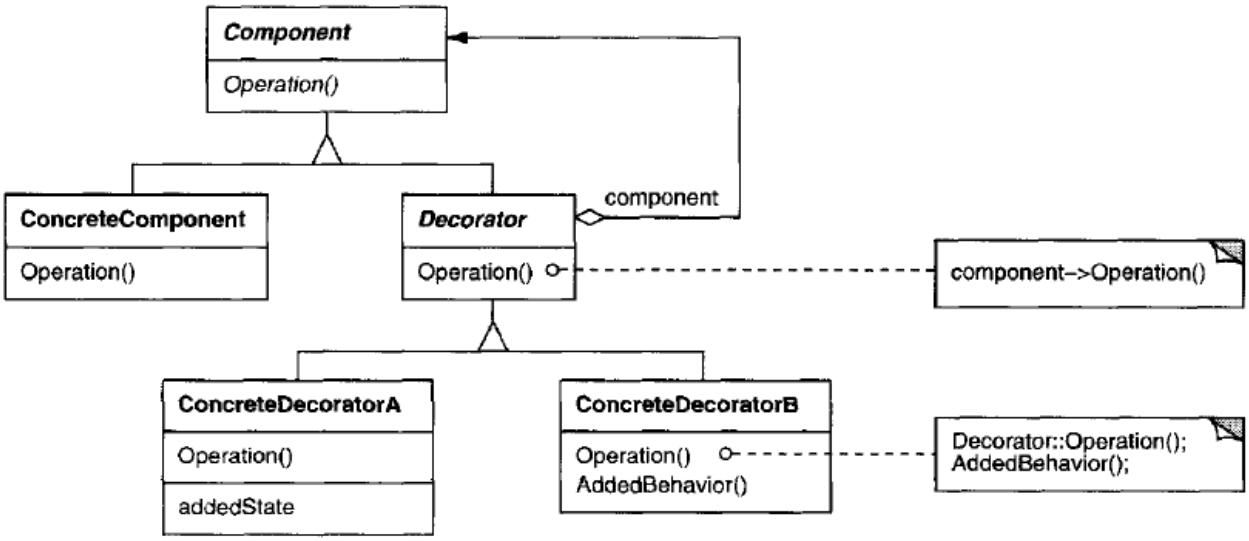
- потрібна можливість за потреби ігнорувати те, що композиції із об’єктів та окремі об’єкти є різними речами та проводити над ними однакові операції.
- потрібно представити ієрархії об’єктів у вигляді «частина-ціле».

4. Декоратор (Decorator).

   Надає можливість динамічно додавати об’єкту нові обов’язки, доповнюючи новим станом чи поведінкою, через загортання їх у певні обгортки.

   Декоратор є патерном рівня об’єктів.

   Структура:



(Рисунок 2.4 – загальна структура патерну Декоратор [1])

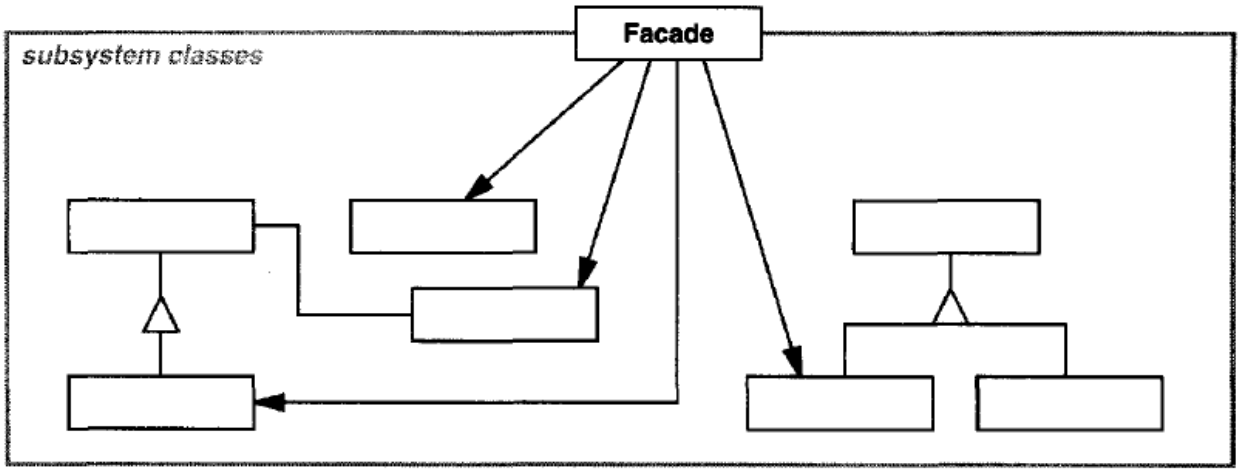
Патерн використовується, коли:

- потрібно динамічно додавати та забирати певні обов’язки у об’єктів.
- потрібно передбачити багато комбінацій можливих станів системи, і розширення через наслідування стає непрактичним через надто громіздку ієрархію та часте дублювання коду.

5. Фасад (Facade).

   Забезпечує наявність одного уніфікованого інтерфейсу високого рівня до певної кількості інших інтерфейсів у підсистемі. Фасад є патерном рівня об’єктів.

   Структура:



(Рисунок 2.5 – загальна структура патерну Фасад [1])

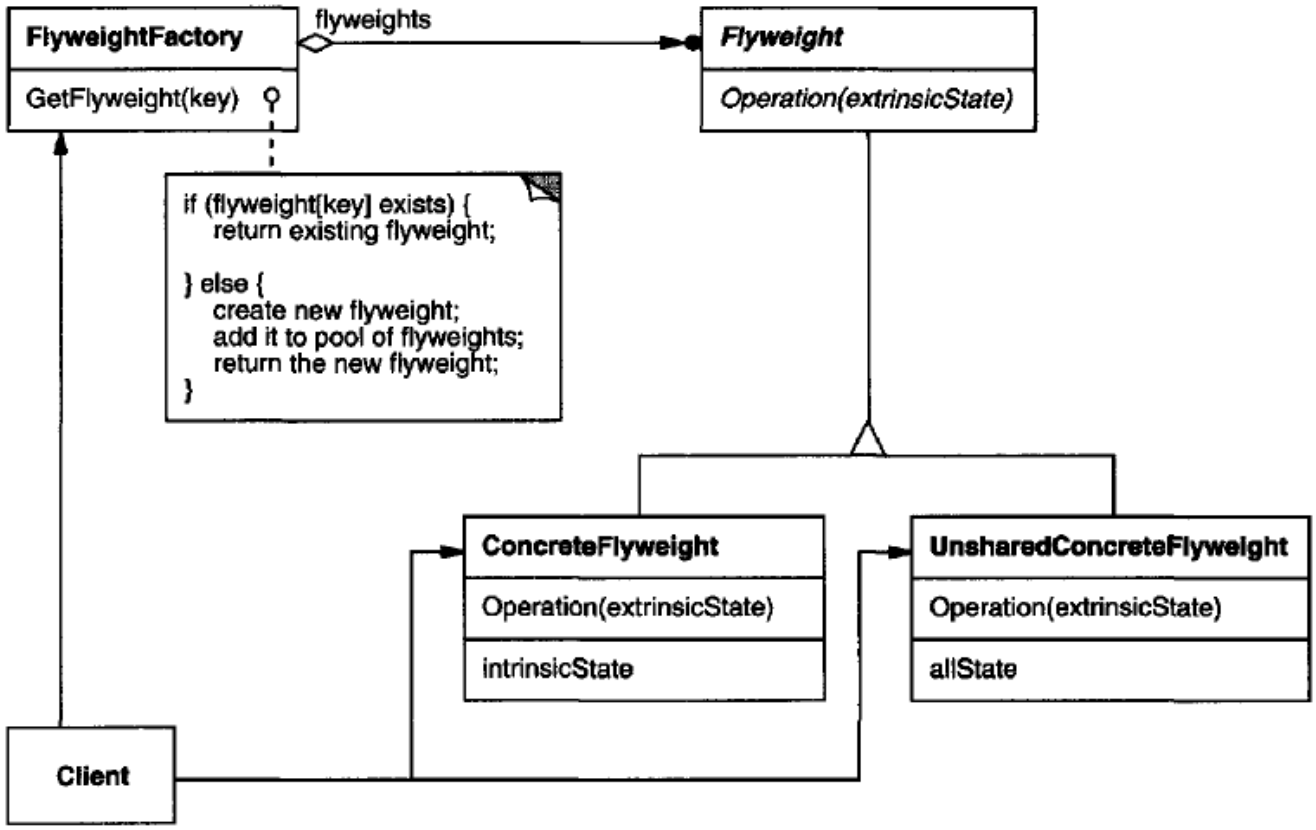
Патерн використовується, коли:

- потрібно полегшити використання складної підсистеми, надаючи клієнтам простий інтерфейс для часто повторюваних операцій, що рідко потребують кастомізації.
- потрібно надати один високорівневий інтерфейс, щоб сховати або замінити багато низькорівневих інтерфейсів.
- потрібно розбити систему на рівні та надати одну уніфіковану точку входу на рівень.

6. Легковаговик (Flyweight).

   Дозволяє більш ефективно зберігати об’єкти, заощаджуючи ресурс оперативної пам’яті. Об’єкти, що мають однаковий стан, не зберігають у собі зайві дані. Легковаговик є патерном рівня об’єктів.

   Структура:



(Рисунок 2.6 – загальна структура патерну Легковаговик [1])

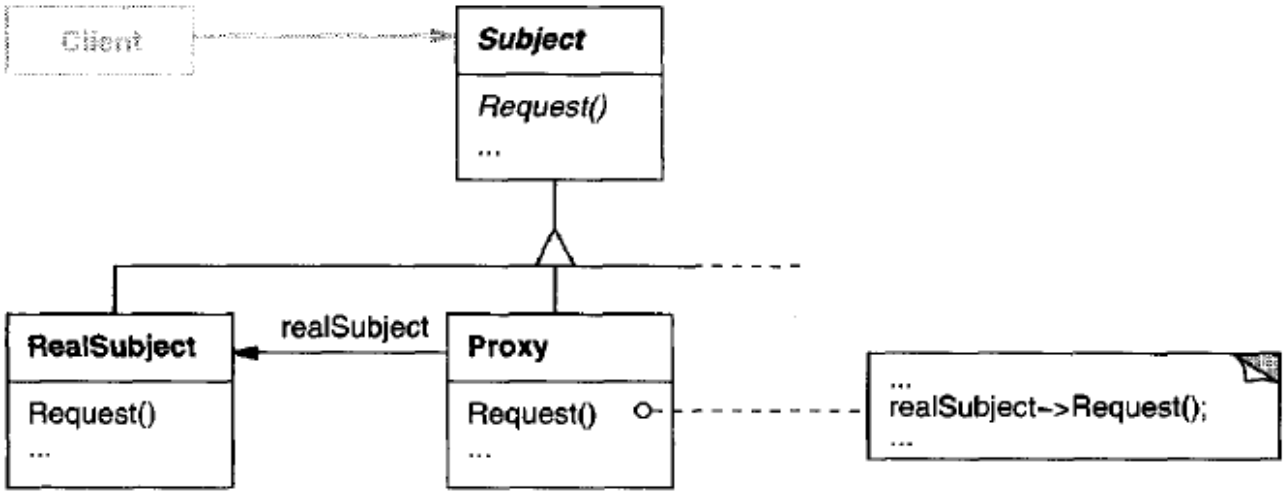
Патерн використовується, коли:

- система оперує великою кількістю об’єктів.
- потрібно особливо дбайливо ставитись до ресурсу фізичної пам’яті та одночасно унікальність та точна ідентифікація об’єктів є неважливими.

7. Заступник (Proxy).

   Забезпечує наявність об’єкта-заступника для контролю доступу до іншого об’єкту. Заступник є патерном рівня об’єктів.

   Структура:



(Рисунок 2.7 – загальна структура патерну Заступник [1])

Патерн використовується, коли:

- потрібно скрити реальне розташування об’єкта в пам’яті від клієнта (віддалений замісник).
- Потрібно створювати ресурсоємні об’єкти за запитом, реалізуючи відкладену ініціалізацію (віртуальний замісник).
- Дозволяє проводити додаткові операції, при спробі доступитись до об’єкту . Наприклад, верифікації або валідацію (захисний замісник)

2.3 Приклад компонування кількох структурних патернів в одну систему

Завдання: спроектувати систему, яка повертає найбільш точні дані про погоду (детально на найближчі три доби, загально на тиждень, загально на місяць) для певного обраного міста у певній країні. Для отримання найбільш точної інформації про погодні умови певного регіону, пропонується використовувати дані, що надаються регіональним гідрометцентром.

Постановка задачі проектованої системи: від системи потребується вичерпний набір простих операцій, що завжди повертають дані у уніфікованому вигляді. Проте, для отримання точних даних потребується використовувати багато сторонніх ресурсів, що мають різноманітні інтерфейси.

При проектуванні такої системи доцільно скористатися патернами Фасад та Адаптер.

2.3.1 Патерн фасад (facade)

Роль патерну Фасад у системі: забезпечує єдиний простий інтерфейс, скриваючи використання багатьох підсистем, що приймають участь у генеруванні даних.

Приклади використання патерну Фасад у реальних системах:

Фасад	
Фреймворк	Використання
.NET Framework	Наприклад MessageBox (System.Windows.Forms) - забезпечує багаторазовий інтерфейс високого рівня для деяких інтерфейсів Windows Forms, замість того, щоб писати багато коду для представлення діалогового вікна, яке ви можете просто написати MessageBox.Show("Hello world");.

(Таблиця 2.1 – використання патерну Фасад у реальних системах)

2.3.2 Патерн адаптер (adapter)

Роль патерну Адаптеру у системі: робить сумісними інтерфейси, що розроблялися третіми особами з уніфікованим інтерфейсом нашої системи.

Приклади використання патерну Адаптер у реальних системах:

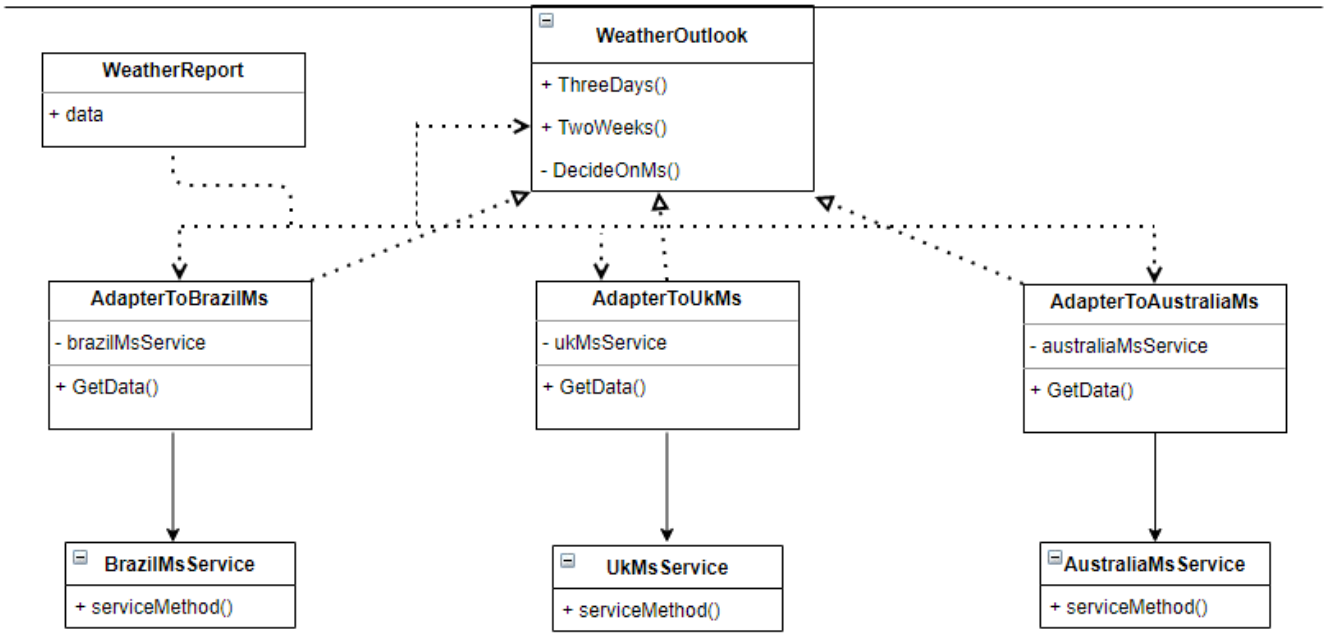
Адаптер	
Фреймворк	Використання
.NET Framework	Адаптація даних. Наприклад клас DataAdapter у .NET Framework

	представляє набір команд даних для підключення до бази даних
Spring Framework	Spring Framework використовує адаптери JMS для надсилання та отримання повідомлень JMS, а адаптери JDBC для перетворення повідомлень у запити до бази даних і повернення наборів результатів до повідомлень.

(Таблиця 2.2 – використання патерну Адаптер у реальних системах)

2.3.3 Компонування патернів у систему

При вирішенні заданої задачі за допомогою патернів Фасад та Адаптер була спроектована така система:



2.4 Висновок

У даному розділі були проаналізовані структурні патерни проектування. Особлива увага була приділена патернам Адаптер та Фасад. Була розглянута доцільність поєднання цих двох патернів при вирішенні конкретної задачі. Як проілюстрованому в нашому прикладі, це поєднання може бути особливо корисним, коли нам потрібно організувати роботу складних підсистем.

Отже, можна зробити висновок, що при використанні одного структурного патерну треба пам’ятати про можливість поєднання його з іншим патерном цього типу.

Розділ 3. Поведінкові патерни

3.1 Призначення поведінкових патернів

Поведінкові патерни займаються алгоритмами і присвоєнням відповідальності об'єктам. Поведінкові патерни описують не тільки самі об'єкти або класи, а й моделі взаємодії між ними. Ці моделі характеризують складний потік управління, за яким важко встежити під час часу виконання.

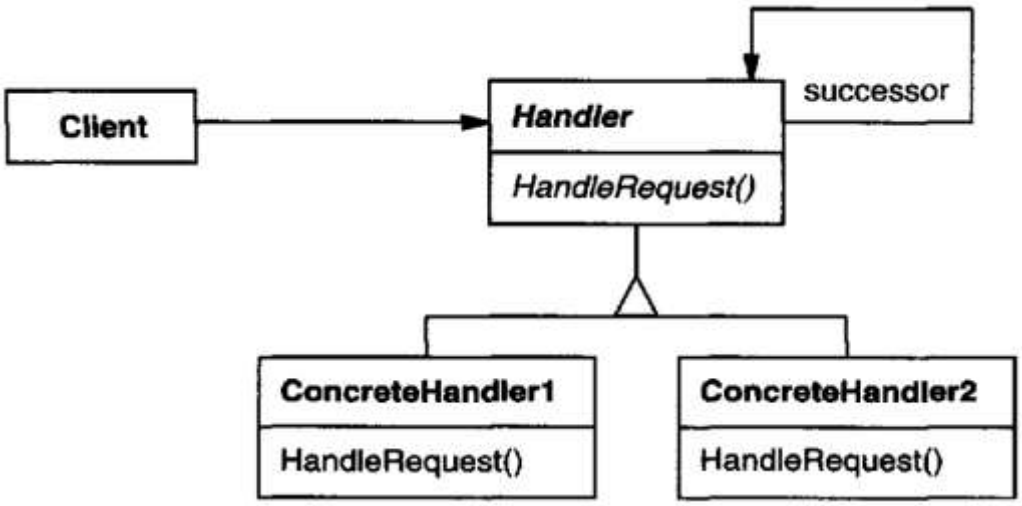
Патерни рівня класів використовують наслідування для розподілу поведінки між класами. Патерни рівня об’єктів використовують композицію.

Використання поведінкових патернів проектування дозволяють розробнику приділяти більше уваги тому, як об'єкти взаємопов'язані, забираючи необхідність концентруватись на потоку управління.

3.2. Характеристика існуючих поведінкових патернів

Існує 11 поведінкових патернів:

1. Ланцюжок обов’язків (Chain of Responsibility).
   Дозволяє уникати прив'язки об’єкта-відправника запиту з його об’єктом-одержувачем, дозволяючи обробити цей запит декільком об’єктам. Даний патерн зв’язує в ланцюги об’єкти-одержувачі та передає запит уздовж цього ланцюга, поки один з об’єктів з ланцюга не обробить його.. Ланцюжок обов’язків є патерном рівня об’єктів.
   Структура:



(Рисунок 3.1 – загальна структура патерну Ланцюжок обов’язків [1])

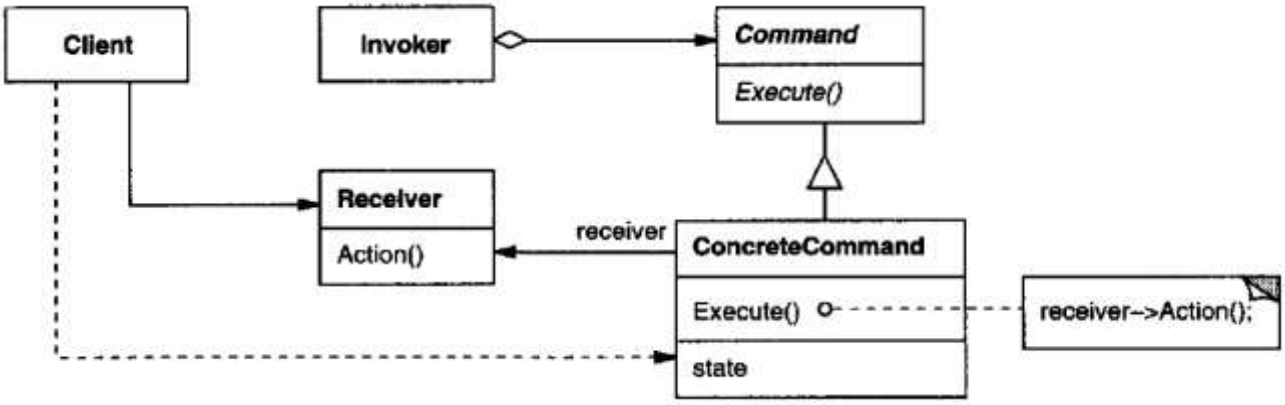
Патерн використовується, коли:

- більше ніж один об'єкт може обробляти запит, і обробник невідомий завчасно. Обробника слід встановлювати автоматично.
- надіслати запит одному з декількох об'єктів, не вказуючи приймач явно.
- набір об'єктів, які можуть обробляти запит, повинен вказуватися динамічно.

2. Команда (Command).

Дозволяє інкапсулювати запит у вигляді об’єкту, та дозволяючи клієнту задавати параметри для обробки цього запиту. Команда є патерном рівня об’єктів.

Структура:



(Рисунок 3.2 – загальна структура патерну Команда [1])

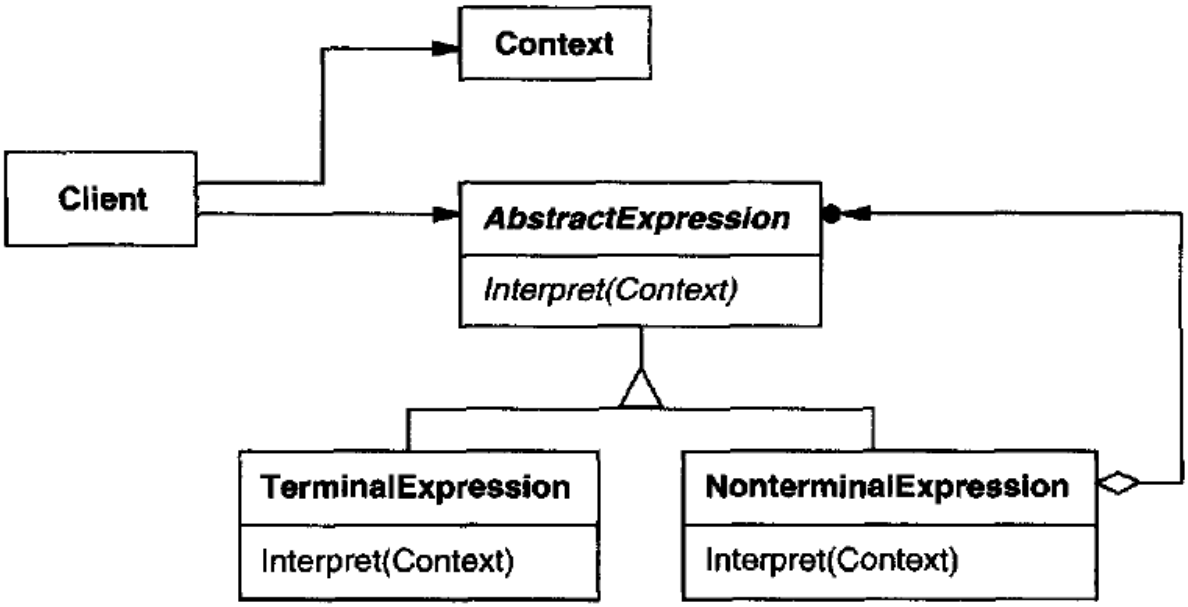
Патерн використовується, коли:

- запит повинен бути виконаним, але необов'язково негайно, якщо у певний момент часу є більш пріоритетні задачі. Тобто запит, перетворений у об'єкт-команду, буде додано до черги задач та буде виконано незалежно від часу оригінального запиту.
- потрібно мати можливість відкликати/відмінити операцію.
- потрібно побудувати систему навколо високорівневих операцій, що базуються на примітивних операціях.

3. Інтерпретатор (Interpreter).

Дозволяє сформувати об'єктно-орієнтовне представлення граматики для заданої мови, а також описує правила створення механізму інтерпретації речень цієї мови. Інтерпретатор є патерном рівня класів.

Структура:



(Рисунок 3.3 – загальна структура патерну Інтерпретатор [1])

Патерн використовується, коли:

Є мова для перекладу, і ви можете представляти твердження в мові як абстрактні дерева синтаксису.

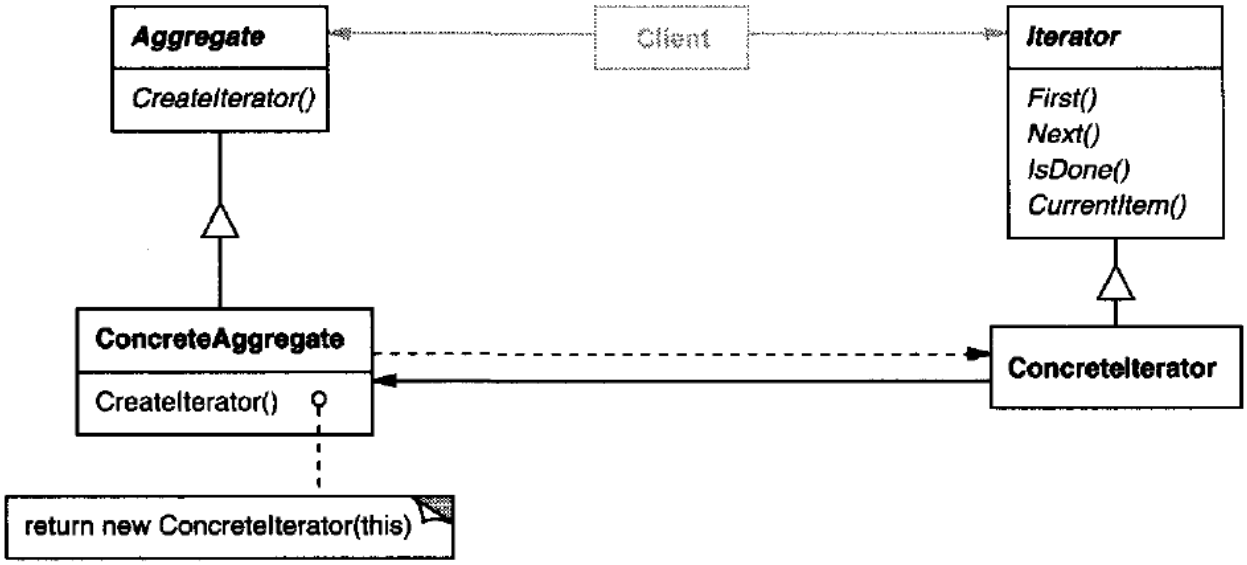
- граматика є простою. Для складних граматик ієрархія класів стає великою і некерованою. Такі інструменти, як генератор синтаксичного аналізатора є кращою альтернативою в таких випадках. Вони можуть інтерпретувати вирази без побудов дерев абстрактних синтаксисів, що може заощадити простір і, можливо, час;
- ефективність не є важливою проблемою. Зазвичай найефективніші інтерпретатори не реалізується шляхом безпосередньої інтерпретації дерев синтаксичного аналізу, а шляхом першого перекладу їх в іншу форму. Наприклад, регулярні вирази часто трансформуються в автомати стану. Але навіть тоді перекладач може бути реалізований за

допомогою даного патерну.

4. Ітератор (Iterator).

   Надає зручний та безпечний спосіб доступу до елементів колекції (складного об’єкту), при цьому не розкриваючи внутрішнього представлення цієї колекції. Ітератор є патерном рівня об'єктів.

   Структура:



(Рисунок 3.4 – загальна структура патерну Ітератор [1])

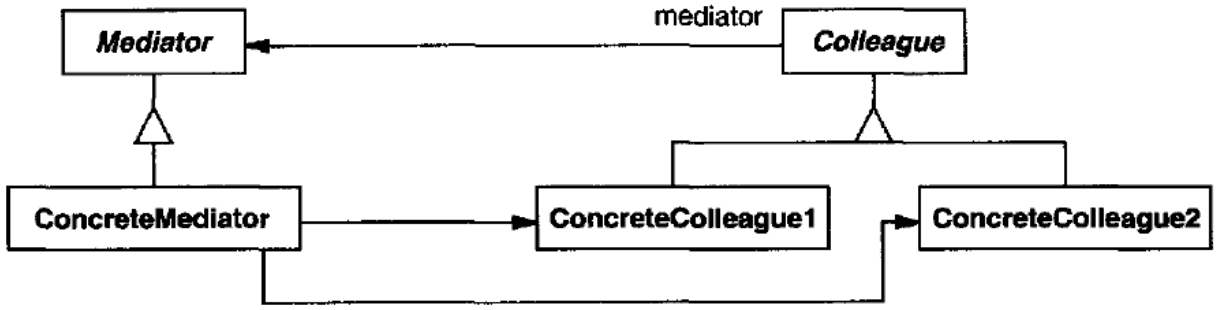
Патерн використовується, коли:

- потрібно отримати доступ до вмісту елементів колекції без розкриття його внутрішнього представлення.
- необхідно створити декілька варіантів обходу колекції.
- потрібно сформувати єдиний інтерфейс(набір методів) для уніфікованого звернення до елементів з різноманітних колекцій, тобто для підтримки поліморфної ітерації (один ітератор для декількох колекцій).

5. Посередник (Mediator).

   Визначає об’єкт, що інкапсулює в собі поведінку множини інших об’єктів. Патерн дозволяє уникнути створення зайвих залежностей між об’єктами для регулювання поведінки системи, через створення єдиного об’єкту-медіатора. Посередник є патерном рівня об'єктів.

   Структура:



(Рисунок 3.5 – загальна структура патерну Посередник [1])

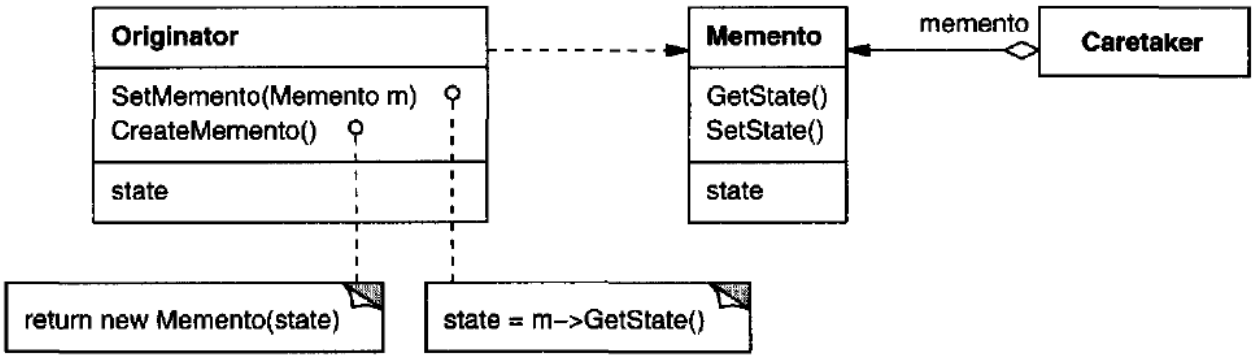
Патерн використовується, коли:

- множина об’єктів має чітко визначену, але складну поведінку об’єктів в середині себе, що призводить до накопичення взаємозалежностей між об’єктами. Що у свою чергу, робить керування системою ще більш складним.
- повторне використання об’єктів у іншому блоці коду стає складним, бо ці об’єкти надто залежать від стану інших об’єктів.
- потрібно розподілити певну поведінку між декількома класами, не створюючи складної ієрархії.

6. Хранитель (Memento).

   Не порушуючи інкапсуляцію, фіксує та виносить за межі об’єкта-власника його внутрішній стан так, щоб винесений стан міг бути відновленим в вихідному об’єкті-власнику потім. Хранитель є патерном рівня об'єктів.

   Структура:



(Рисунок 3.6 – загальна структура патерну Хранитель [1])

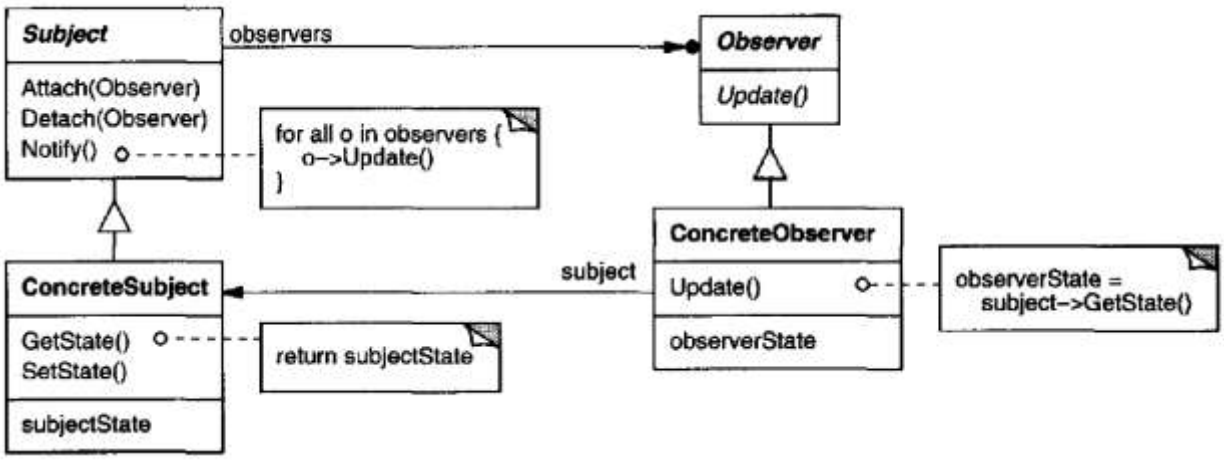
Патерн використовується, коли:

- необхідно зробити фіксацію стану або деякої частини стану об’єкта, щоб він міг бути відновленим до такого стану пізніше.
- потрібно створити прямий інтерфейс для отримання стану розкриє деталі реалізації і зупинить інкапсуляцію об’єкта.

7. Спостерігач (Observer).

   Визначає залежність між об’єктами один-до-багатьох для того, щоб при зміні стану одного об’єкту, інші отримували повідомлення про зміну та змінювали свій стан відповідно. Спостерігач є патерном рівня об'єктів.

   Структура:



(Рисунок 3.7 – загальна структура патерну Спостерігач [1])

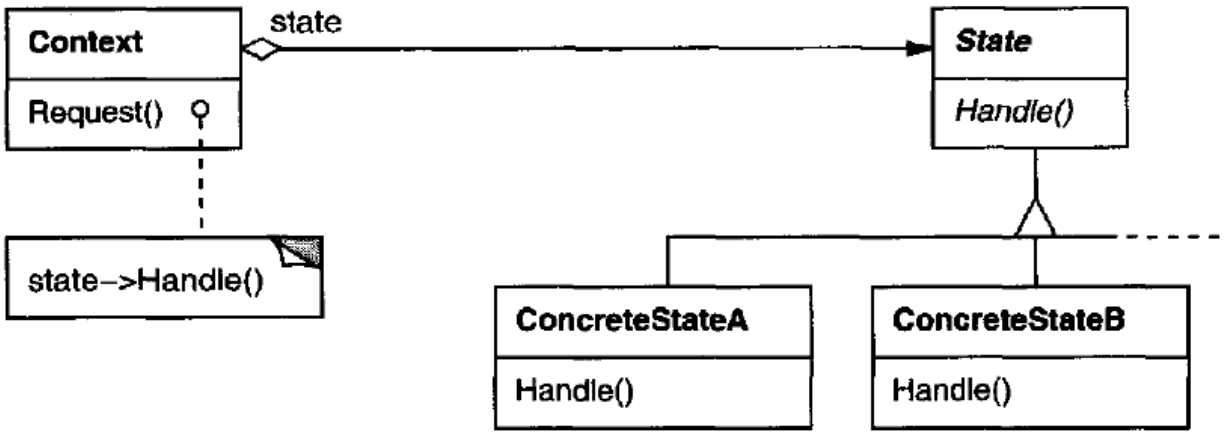
Патерн використовується, коли:

- певні зміни у стані об’єкту повинні ставати автоматично відомими іншим об’єктам, кількість яких заздалегідь невідома.
- у системі є потреба у механізмі повідомлень, проте між об’єктами не повинно бути сильної залежності.
- об’єкту, що змінює свій стан не потрібна інформація про інші об’єкти, що слідкують за змінами його стану.

8. Стан (State)

Дозволяє об’єкту змінювати свою поведінку, в залежності від його стану. Поведінка об'єкту змінюється настільки, що з'являється враження, що змінився клас об'єкту. Стан є патерном рівня об'єктів.

Структура:



(Рисунок 3.8 – загальна структура патерну Стан [1])

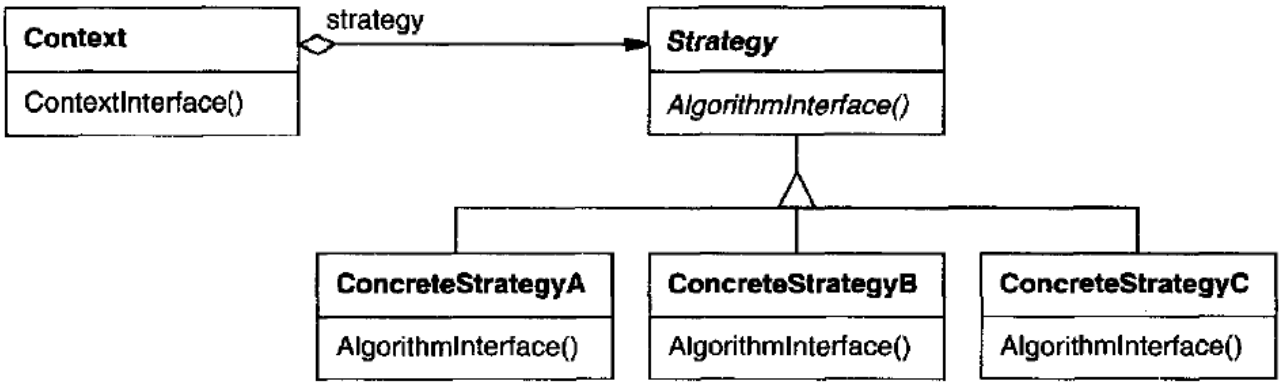
Патерн використовується, коли:

- поведінка об'єкта залежить від його стану, і він повинен змінити свою поведінку під час виконання залежно від цього стану.
- операції мають громіздкі багаточастинні умовні оператори, які залежать від стану об’єкта. Цей стан зазвичай представлений одним або кількома переліченими константами. Часто кілька операцій містять ту саму умовну структуру. Патерн Стан представляє кожну гілку в окремим класом. Це дозволяє вам розглядати стан об’єкта як самостійний об’єкт, який може змінюватися незалежно від інших об’єктів.

9. Стратегія (Strategy).

   Визначає набір алгоритмів, інкапсулює кожен з них в окремий клас і робить їх взаємозамінними. Стратегія дозволяє підміняти алгоритми без участі клієнтів, які використовують ці алгоритми. Стратегія є патерном рівня об'єктів.

   Структура:



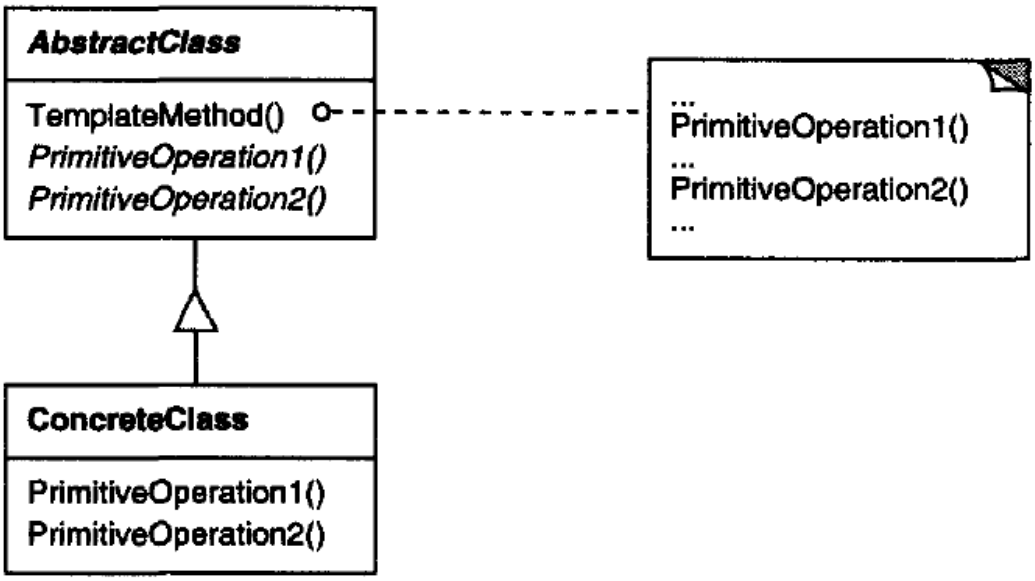
(Рисунок 3.9 – загальна структура патерну Стратегія [1])

Патерн використовується, коли:

- існує декілька класів які відрізняються лише своєю поведінкою. Патерн дозволяє винести кожну з поведінок в окремий клас-стратегію, що будуть використовуватись в залежності від контексту.
- алгоритм використовує дані, про які клієнти не повинні знати. Даний шаблон допоможе уникнути викриття складних структур даних, що залежать від вибраного алгоритму поведінки.
- поведінка класу залежить від великої кількості умов, що ускладнює подальшу взаємодію з цим класом. Замість великої кількості умовних конструкцій можна визначити декілька стратегій.

10. Метод шаблонів (Template Method).

Формує структуру алгоритму і дозволяє в похідних класах реалізувати, замістити або перевизначити певні кроки алгоритму, не змінюючи структури алгоритму загалом. Метод шаблонів є патерном рівня класів. Структура:



(Рисунок 3.10 – загальна структура патерну Метод шаблонів [1])

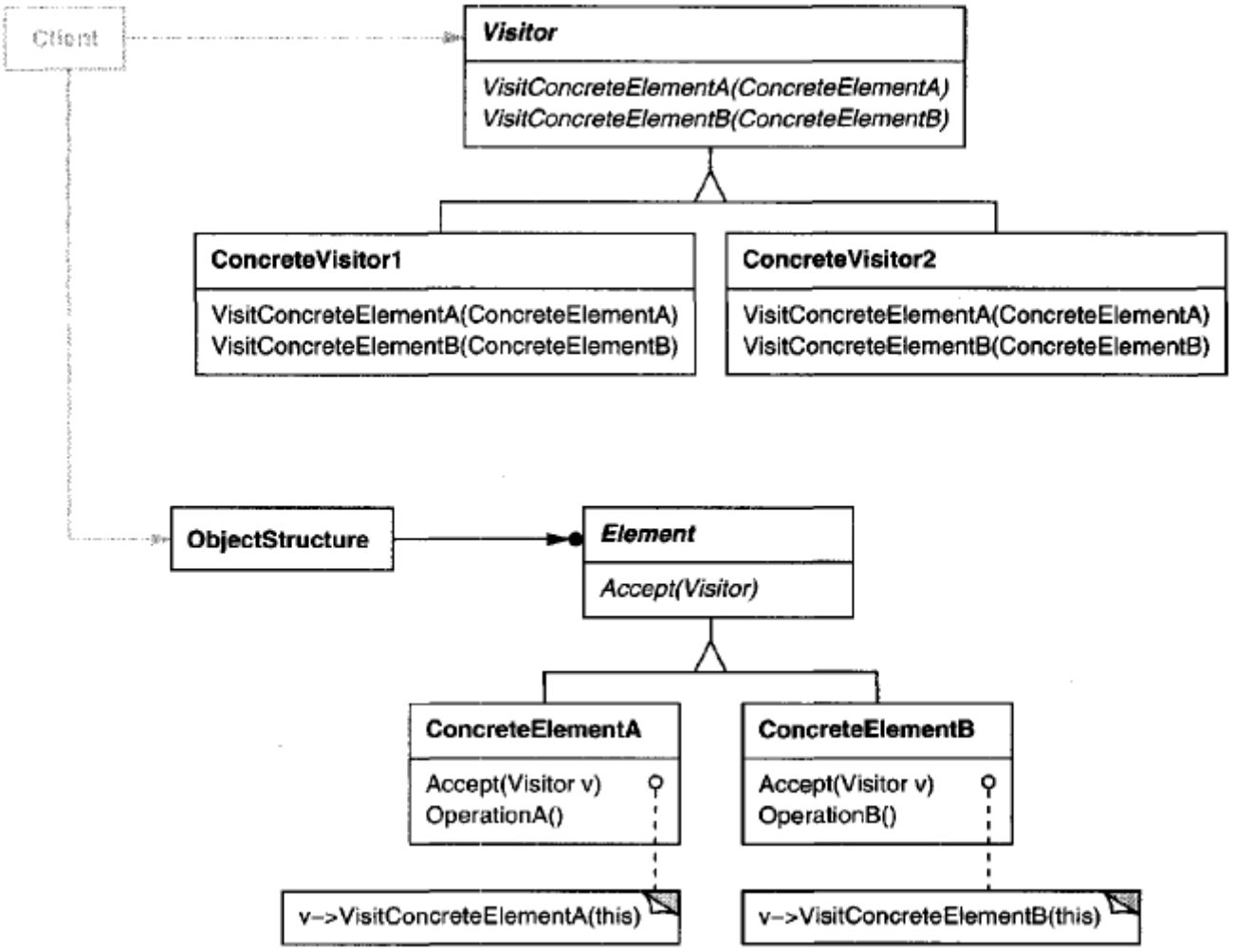
Патерн використовується, коли:

- потрібно в базовому класі організувати незмінну послідовність кроків алгоритму, дозволяючи змінювати реалізацію кожного окремого кроку в похідних класах;
- необхідно в базовому класі описати послідовність кроків алгоритму, спільну для всіх підкласів, з метою уникнути дублювання коду.
- для управління розширеннями підкласів. Ви можете визначити метод шаблону, який викликає певні операції у конкретних точках, тим самим дозволяючи розширення лише в цих точках.

11. Відвідувач (Visitor).

Дозволяє одноманітно обійти набір елементів з різними інтерфейсами (тобто набір об'єктів різних класів не приводячи їх спільного базового типу), а також дозволяє додати новий метод в клас об'єкта, при цьому не змінюючи сам клас цього об'єкта. Відвідувач є патерном рівня об'єктів.

Структура:



(Рисунок 3.11 – загальна структура патерну Відвідувач [1])

Патерн використовується, коли:

- структура даних містить багато об'єктів різних типів з різними інтерфейсами, та є потреба виконувати схожі операції з кожним із них.
- потрібно спростити додавання нових операцій у роботі зі складними структурами об’єктів.

3.3 Приклад компонування кількох поведінкових патернів в одну систему

Завдання: спроектувати бот, що в залежності поточного курсу двох валют на біржі вирішує розмістити або прибрати ордер на купівлю валюти. Для проведення транзакції потрібно зменшити відповідний баланс у локальному сховищі на суму транзакції, для того щоб інші боти не могли провести операції на суму, якої немає.

Дана система повинна відслідковувати стан (processed, pending, declined) розміщених ордерів на біржі, при зміні стану ордеру та в залежності від його стану, бот повинен виконати певні дії у локальному сховищі. А саме:

processed: змінити стан ордеру у локальному сховищі.

pending: не робити нічого.

declined: додати суму транзакції назад у баланс.

Постановка задачі проектованої системи: від системи потребується реалізувати механізм автоматичних нотифікацій, при зміні стану певного об’єкту. Бажано інкапсулювати набір команд у об’єктах для полегшення відслідковування потоку управління

При проектуванні такої системи доцільно скористатися патернами Команда та Спостерігач.

3.3.1 Патерн команда (command)

Роль патерну Команда у системі: інкапсулювати набір команд у об’єктах для полегшення відслідковування потоку управління.

Приклади використання патерну Команда у реальних системах:

Команда	
Фреймворк	Використання
Django	В компоненті фреймворка HttpRequest. За допомогою нього запит інкапсулюється в об'єкті
Spring Framework	Часто зустрічається в GUI, наприклад, обробка подій кнопок, пунктів меню, посилань, а також індикаторів прогресу Java

(Таблиця 3.1 – використання патерну Команда у реальних системах)

3.3.2 Патерн спостерігач (observer)

Роль патерну Спостерігач у системі: відслідковувати стан кожного об’єкту певного типу, сповіщаючи про зміни інші частини системи.

Приклади використання патерну Спостерігач у реальних системах:

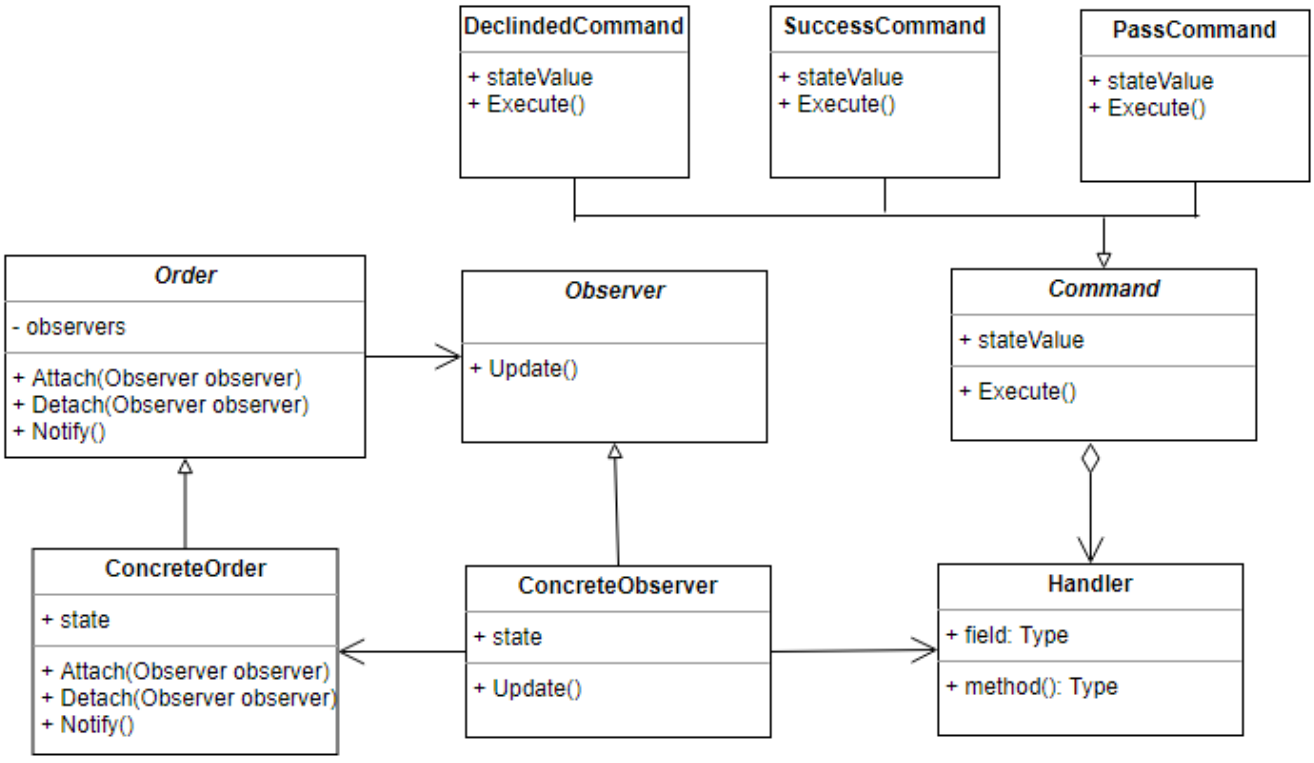
Спостерігач	
Фреймворк	Використання
Django	В компоненті фреймворка Signals. Коли один об'єкт змінює стан, всі його слухачі отримують повідомлення та оновлюються автоматично

.NET Framework	В .NET даний патерн підтримується двома способами: за допомогою інтегрованих у мову подій (тісно пов'язані спостерігачі) та через інтерфейси IObservable / IObserver

(Таблиця 3.1 – використання патерну Команда у реальних системах)

3.3.3 Компонування патернів у систему

При вирішенні заданої задачі за допомогою патернів Фасад та Адаптер була спроектована така система:



(Рисунок 3.3.3.1 – спроектована система за сформульованим завданням)

3.4 Висновок

У даному розділі були проаналізовані структурні патерни проектування. Особлива увага була приділена патернам Команда та Спостерігач. В цьому випадку поєднання патернів може бути особливо корисним, коли ми зустрічаємось з необхідністю організовувати систему, що має багато комбінацій станів та потік управління якої є досить складним для відстежування.

Отже, можна зробити висновок, що при використанні одного поведінкового патерну треба пам’ятати про можливість поєднання його з іншим патерном цього типу.

Висновки

В даній курсовій роботі було проаналізовано стандартні патерни проектування і досліджено доцільність їх поєднання при вирішенні конкретних задач.

Була досліджена теорія використання кожного патерну, оскільки перед використанням патерну важливо розуміти, яку проблему він повинен вирішувати за своїм типом призначення. Особливу увагу було приділено 6 патернам: абстрактній фабриці, одинаку, адаптеру, фасаду, команді та спостерігачу. Також, було спроектовано 3 системи, що демонструють поєднання використання патернів одного типу.

В розділі 1 були проаналізовані породжуючі патерни проектування. Більш детально були розглянуті патерни Абстрактна фабрика та Одинак. Була розглянута доцільність поєднання цих двох патернів на конкретному прикладі. Висновок - поєднання породжуючих патернів може зробити систему більш надійною, гнучкою та покращити якість розробленої системи.

В розділі 2 були проаналізовані структурні патерни проектування. Була розглянута доцільність поєднання цих двох патернів при вирішенні конкретної задачі. Висновок - поєднання структурних патернів може бути особливо корисним, коли нам потрібно організувати роботу складних підсистем.

В розділі 3 були проаналізовані структурні патерни проектування. Більш детально були розглянуті патерни Команда та Спостерігач. Досліджене на конкретному прикладі поєднання цих двох патернів показало, що у випадку, коли є необхідність організовувати систему з багатьма комбінаціями станів і потік управління якої є досить складним для відстеження, таке поєднання може бути дуже корисним.

Отже, протягом цієї роботи було розглянуто можливості одночасного використання кількох патернів одного типу за призначенням. Був зроблений основний висновок про те, що при використанні будь-якого патерну треба неодмінно пам’ятати про можливість поєднання його з іншим патерном цього типу. Це дає можливість зробити систему більш надійною, гнучкою, покращити її якість, прозорість, спростити організацію роботи складних підсистем, спростити складний потік управління.

Список використаної літератури

1. Design Patterns: Elements of Reusable Object-Oriented Software (1994) by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.
2. Head First Design Patterns: A Brain-Friendly Guide by Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra.
3. Agile Software Development, Principles, Patterns, and Practices by Robert Cecil Martin.
4. Design Patterns via C#: Олександр Шевчук, Дмитро Охріменко, Олексій Касьянов.
5. Design Patterns [Електронний ресурс]. – режим доступу: https://sourcemaking.com/design_patterns
6. Design Patterns [Електронний ресурс]. – режим доступу: https://refactoring.guru