1. Базові відомості про патерни проектування та їхнє використання.

1.1 Що таке патерн

**Патерн проектування — це вирішення серйозних проблем при проектуванні архітектури програмного забезпечення, що часто зустрічається.**

На відміну від готових функцій або бібліотек, патерн не можна просто взяти та скопіювати у програму. Патерн є не якимось конкретним кодом, а загальною концепцією вирішення тієї чи іншої проблеми, яку потрібно буде ще корегувати під потреби вашої програми.

Об'єктно-орієнтовані шаблони показують відносини та взаємодії між класами чи об'єктами, без визначення того, які кінцеві класи чи об'єкти програми використовуватимуться.

Патерни часто плутають із алгоритмами, адже обидва поняття описують типові рішення якихось відомих проблем. Але якщо алгоритм – це чіткий набір дій, то патерн — це високорівневий опис рішення, реалізація якого може відрізнятись у двох різних програмах.

Якщо навести аналогії, то алгоритм – це кулінарний рецепт із чіткими кроками, а патерн – інженерний креслення, на якому намальовано рішення, але не конкретні кроки його реалізації.

"Низькорівневі" шаблони, що враховують специфіку конкретної мови програмування, називаються ідіомами. Це хороші рішення проектування, характерні для конкретної мови чи програмної платформи, і тому не універсальні.

Найуніверсальніші — архітектурні патерни, які можна реалізувати практично будь-якою мовою. Вони потрібні для проектування всієї програми, а не окремих її елементів.

Описи патернів зазвичай дуже формальні і найчастіше складаються з таких пунктів:

* проблема, яку вирішує патерн;
* мотивації до вирішення проблеми у спосіб, який пропонує патерн;
* структури класів, що становлять рішення;
* приклад на одній з мов програмування;
* особливостей реалізації у різних контекстах;
* зв'язків із іншими патернами.

1.2 Історія патернів

У 1970-х роках архітектор Крістофер Олександр склав набір шаблонів проектування. У галузі архітектури ця ідея не отримала такого розвитку, як пізніше у сфері програмної розробки.

У 1987 році Кент Бек (Kent Beck) і Вард Каннінгем (Ward Cunningham) взяли ідеї Олександра і розробили шаблони стосовно розробки програмного забезпечення для розробки графічних оболонок мовою Smalltalk.

В 1988 Еріх Гамма (Erich Gamma) почав писати докторську дисертацію при цюріхському університеті про загальну переносимість цієї методики на розробку програм.

У 1989-1991 роках Джеймс Коплін (James Coplien) працював над розробкою ідіом для програмування на C++ та опублікував у 1991 році книгу Advanced C++ Idioms.

Цього ж року Еріх Гамма закінчує свою докторську дисертацію та переїжджає до США, де у співпраці з Річардом Хелмом (Richard Helm), Ральфом Джонсоном (Ralph Johnson) та Джоном Вліссідесом (John Vlissides) публікує книгу Design Patterns — Elements ofus Software. У цій книзі описано 23 шаблони проектування. Також команда авторів цієї книги відома загалу під назвою «Банда чотирьох» (англ. Gang of Four, часто скорочується до GoF). Саме ця книга стала причиною зростання популярності шаблонів проектування.

1.3 Вигоди від використання шаблонів:

* Застосування багатьох шаблонів проектування дозволяє створювати більш гнучке та модифіковане програмне забезпечення. Причина у тому, що ці рішення вже випробувані часом. Тому використання шаблонів дозволяє створювати структури, що допускають їх модифікацію більшою мірою, ніж це можливо у разі рішення, що першим спало на думку.
* Шаблони проектування, вивчені належним чином, значно допомагають загальному розумінню основних принципів об'єктно-орієнтованого проектування.
* Шаблони проектування дозволяють розробнику знаходити проектні рішення для складних проблем, не створюючи громіздкої ієрархії спадкування класів. Навіть якщо шаблони не використовуються в проекті безпосередньо, тільки зменшення розміру ієрархій успадкування класів вже сприятиме підвищенню якості проекту.
* Загальний словник програмістів, тобто замість того, щоб пояснювати іншим програмістам, як и що робить дані класи, можна назвати патерн і програмісти зрозуміють, за що відповідає даний клас.
  1. Недоліки патернів
* Милиці для слабкої мови програмування. Вперше цей погляд висловив Пол Грем в есе «Помста Ботанів». Потреба в патернах виникає тоді, коли люди вибирають для свого проекту мову програмування з недостатнім рівнем абстракції. В цьому випадку, патерни - це милиця, яка надає цій мові суперздатності. Наприклад, стратегія Strategy в сучасних мовах можна реалізувати простою анонімною (лямбда) функцією.
* Неефективні рішення. Патерни намагаються стандартизувати підходи, які й так вже використовуються. Ця стандартизація здається деяким людям допомогою і вони реалізують патерни «як у книжці», не пристосовуючи патерни до реалій проекту.
* Невиправдане застосування. Схожа проблема виникає у новачків, які тільки-но познайомилися з патернами. Вникнувши в патерни, людина намагається застосувати свої знання скрізь. Навіть там, де можна було б обійтися простіше кодом.
  1. Класифікація патернів

Патерни відрізняються за рівнем складності, деталізації та охоплення проектованої системи. Проводячи аналогію з будівництвом, ви можете підвищити безпеку перехрестя, поставивши світлофор, а можете замінити перехрестя цілою автомобільною розв'язкою з підземними переходами. Крім того, патерни відрізняються призначенням.

Основні типи патернів:

* Поведінкові патерни — дбають про ефективну комунікацію між об'єктами. До них належать: Ланцюг обов’язків (Chain of Responsibility), Команда (Command), Ітератор (Iterator), Посередник (Mediator), Знімок (Memonto), Спостерігач (Observer), Стан (State), Стратегія (Strategy), Шаблонний метод (Template Method), Відвідувач (Visitor).
* Породжуючі патерни — турбуються про гнучке створення об'єктів без внесення в програму зайвих залежностей. До таких належать: Фабричний метод (Factory Method), Абстрактна фабрика (Abstract Factory), Будівельник (Builder), Прототип (Prototype), Одинак (Singleton).
* Структурні патерни показують різні способи побудови зв’язків між об’єктами. В них входять: Адаптер (Adapter), Міст (Bridge), Компонувальник (Composite), Декоратор (Decorator), Фасад (Facade), Легковаговик (Flyweight), Замісник (Proxy).

1.6 Патерн Ланцюг обов’язків (Chain of Responsibility)

Ланцюжок обов’язків — це поведінковий патерн проектування, що дає змогу передавати запити послідовно ланцюжком обробників. Кожен наступний обробник вирішує, чи може він обробити запит сам і чи варто передавати запит далі ланцюжком.

Шаблон рекомендований для використання в умовах:

* в системі, що розробляється, є група об'єктів, які можуть обробляти повідомлення певного типу;
* всі повідомлення мають бути оброблені хоча б одним об'єктом системи;
* повідомлення у системі обробляються за схемою «оброби сам чи перейшли іншому», тобто одні повідомлення обробляються тому рівні, де вони отримані, інші пересилаються об'єктам іншого рівня.

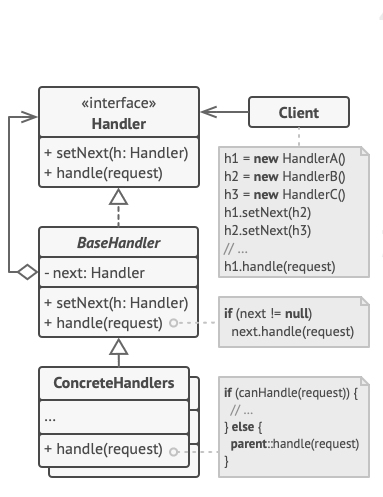


Рисунок 1 – Структура патерна Chain of Responsibility

# 1.7 Патерн Команда (Command)

# Команда — це поведінковий патерн проектування, який перетворює запити на об’єкти, дозволяючи передавати їх як аргументи під час виклику методів, ставити запити в чергу, логувати їх, а також підтримувати скасування операцій.

# Шаблон Команда може бути корисною у таких випадках:

# Кнопки інтерфейсу користувача та пункти меню;

# запис макросів;

# багаторівневе скасування операцій (Undo);

# мережі;

# індикатори виконання;

# пули потоків;

# транзакції;

# майстри.

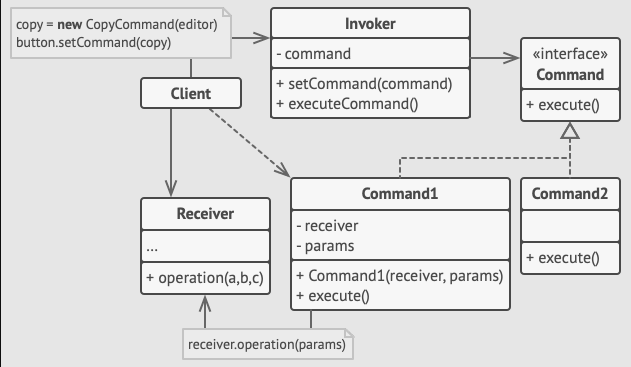


Рисунок 2 – Структура патерна Команда (Command)

1.8 Патерн Ітератор (Iterator)

**Ітератор** — це поведінковий патерн проектування, що дає змогу послідовно обходити елементи складових об’єктів, не розкриваючи їхньої внутрішньої організації.

* Коли необхідно здійснити обхід об'єкта без розкриття його внутрішньої структури
* Коли є набір складових об'єктів, і треба забезпечити єдиний інтерфейс для їхнього перебору
* Коли необхідно надати кілька альтернативних варіантів перебору того самого об'єкта

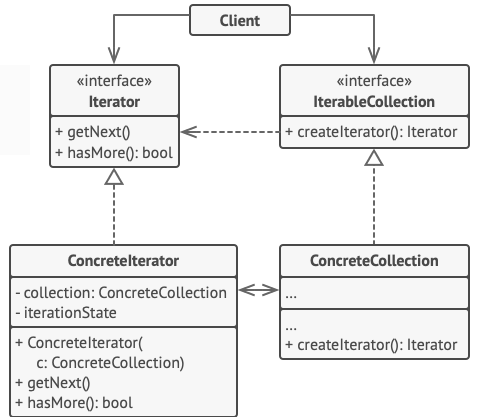


Рисунок 3 – Структура патерна Ітератор (Iterator)

1.8 Патерн посередник (Mediator)

**Посередник**— це поведінковий патерн проектування, що дає змогу зменшити зв’язаність великої кількості класів між собою, завдяки переміщенню цих зв’язків до одного класу-посередника.

Коли використовується патерн:

* коли є безліч взаємозалежних об'єктів, зв'язки між якими складні та заплутані;
* коли необхідно повторно використовувати об'єкт, однак повторне використання утруднено через сильні зв'язки з іншими об'єктами.

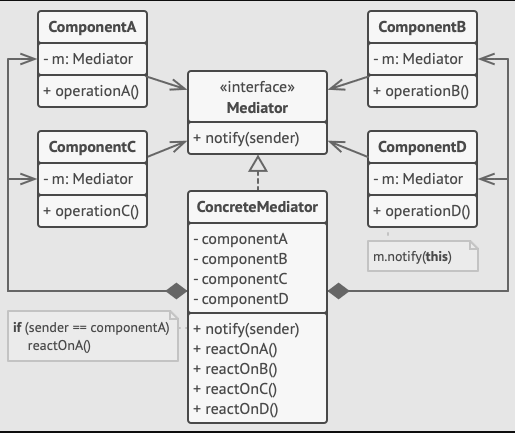


Рисунок 4 – Структура патерна посередник (Mediator)

1.9 Патерн знімок (Memonto)

**Знімок** — це поведінковий патерн проектування, що дає змогу зберігати та відновлювати минулий стан об’єктів, не розкриваючи подробиць їхньої реалізації.

Шаблон Зберігач використовується, коли:

* необхідно зберегти знімок стану об'єкта (або його частини) для подальшого відновлення;
* прямий інтерфейс отримання стану об'єкта розкриває деталі реалізації та порушує інкапсуляцію об'єкта.

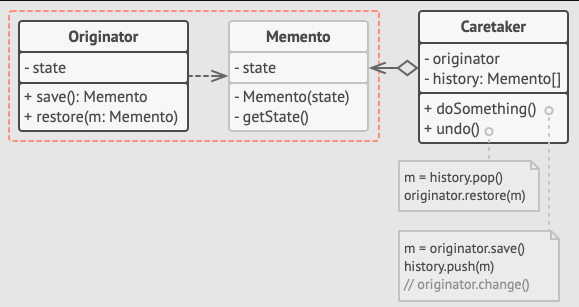


Рисунок 5 – Структура патерна знімок (Memonto)

1.10 Патерн спостерігач (Observer)

**Спостерігач**— це поведінковий патерн проектування, який створює механізм підписки, що дає змогу одним об’єктам стежити й реагувати на події, які відбуваються в інших об’єктах.

Використовувати патерн Спостерігач потрібно, коли:

* система складається з багатьох класів, об'єкти яких повинні знаходитися в узгоджених станах;
* загальна схема взаємодії об'єктів передбачає дві сторони: одна розсилає повідомлення є головним, інша отримує повідомлення і реагує них. Відділення логіки обох сторін дозволяє їх розглядати незалежно та використовувати окремо один від одного;
* існує один об'єкт, який надсилає повідомлення, і безліч передплатників, які отримують повідомлення. При цьому точне число передплатників заздалегідь невідоме і процес роботи програми може змінюватися.

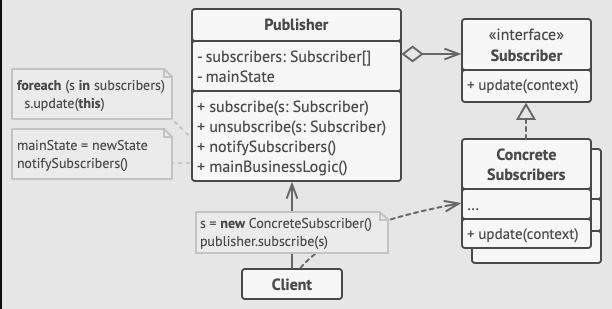


Рисунок 6 – Структура патерна спостерігач (Observer)

1.11 Патерн стан (State)

**Стан** — це поведінковий патерн проектування, що дає змогу об’єктам змінювати поведінку в залежності від їхнього стану. Ззовні створюється враження, ніби змінився клас об’єкта.

Використовувати цей патерн потрібно, коли:

* поведінка об'єкта має залежати від його стану і може динамічно змінюватися під час виконання;
* в коді методів об'єкта використовуються численні умовні конструкції, вибір яких залежить від стану об'єкта.

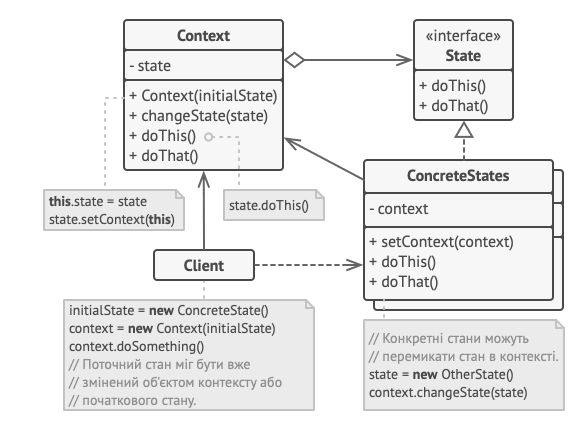


Рисунок 7 – Структура патерна стан (State)

1.12 Патерн стратегія (Strategy)

**Стратегія** — це поведінковий патерн проектування, який визначає сімейство схожих алгоритмів і розміщує кожен з них у власному класі. Після цього алгоритми можна заміняти один на інший прямо під час виконання програми.

Використовувати патерн **Стратегія** потрібно, коли:

* є кілька родинних класів, що відрізняються поведінкою. Можна встановити один основний клас, а різні варіанти поведінки винести в окремі класи і при необхідності їх застосовувати;
* необхідно забезпечити вибір із кількох варіантів алгоритмів, які можна легко змінювати залежно від умов;
* необхідно змінювати поведінку об'єктів на стадії виконання програми;
* клас, який застосовує певну функціональність, нічого не повинен знати про її реалізацію.

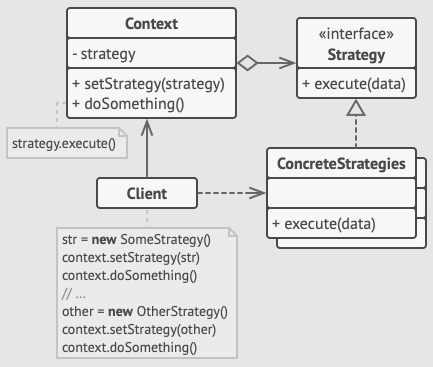


Рисунок 8 – Структура патерна стратегія (Strategy)

1.13 Патерн шаблонний метод (Template Method)

**Шаблонний метод** — це поведінковий патерн проектування, який визначає кістяк алгоритму, перекладаючи відповідальність за деякі його кроки на підкласи. Патерн дозволяє підкласам перевизначати кроки алгоритму, не змінюючи його загальної структури.

Використовувати патерн **Шаблонний метод**  потрібно, коли:

* планується, що в майбутньому підкласи мають перевизначати різні етапи алгоритму без зміни його структури;
* класах, реалізуючим подібний алгоритм, відбувається дублювання коду. Винесення загального коду шаблонний метод зменшить його дублювання в підкласах.

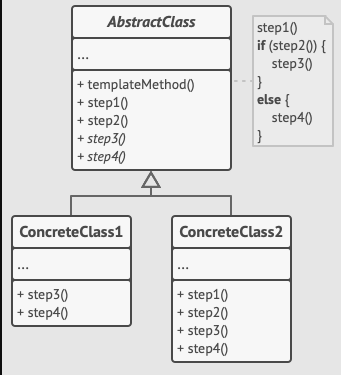


Рисунок 9 – Структура патерна шаблонний метод (Template Method)

1.14 Патерн Відвідувач (Visitor)

**Відвідувач** — це поведінковий патерн проектування, що дає змогу додавати до програми нові операції, не змінюючи класи об’єктів, над якими ці операції можуть виконуватися.

Використовувати патерн Відвідувач потрібно, коли:

* є багато об'єктів різнорідних класів з різними інтерфейсами, потрібно виконати ряд операцій над кожним з цих об'єктів;
* класам необхідно додати однаковий набір операцій без зміни цих класів;
* часто додаються нові операції до класів, загальна структура класів стабільна і практично не змінюється.

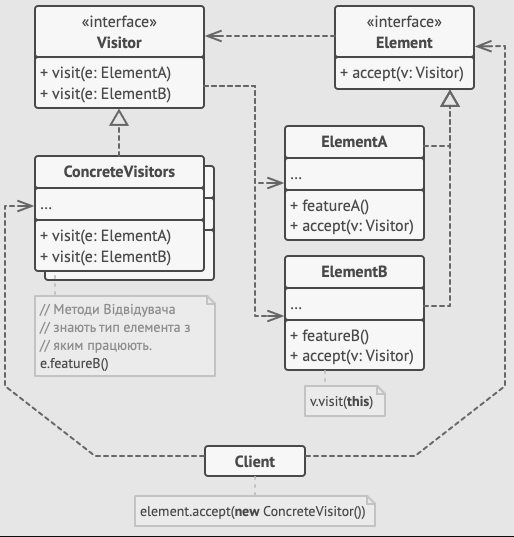


Рисунок 9 – Структура патерна відвідувач (Visitor)

1.15 Патерн Фабричний метод (Factory Method)

**Фабричний метод** — це породжувальний патерн проектування, який визначає загальний інтерфейс для створення об’єктів у суперкласі, дозволяючи підкласам змінювати тип створюваних об’єктів.

Використовувати патерн Фабричний метод потрібно, коли:

* невідомо, об'єкти яких типів необхідно створювати;
* система має бути незалежною від процесу створення нових об'єктів і розширюється: у неї можна легко вводити нові класи, об'єкти яких система має створювати;
* створення нових об'єктів необхідно делегувати з базового класу класам спадкоємцям.

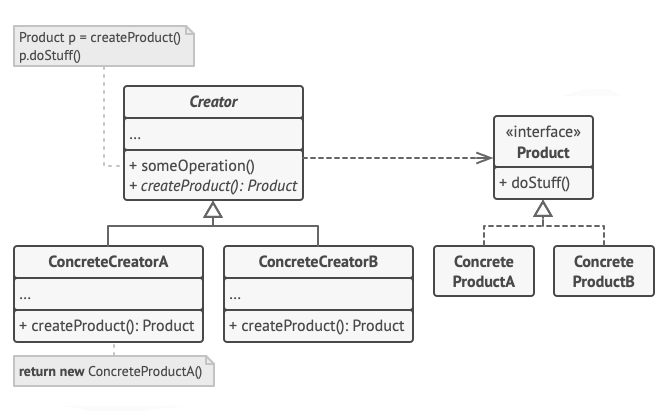


Рисунок 10 – Структура патерна Фабричний метод

1.16 Патерн Абстрактна фабрика (Abstract Factory)

**Абстрактна фабрика** — це породжувальний патерн проектування, що дає змогу створювати сімейства пов’язаних об’єктів, не прив’язуючись до конкретних класів створюваних об’єктів.

Використовувати патерн Абстрактна фабрика потрібно, коли:

* система не повинна залежати від способу створення та компонування нових об'єктів;
* об'єкти, що створюються, повинні використовуватися разом і є взаємопов'язаними.

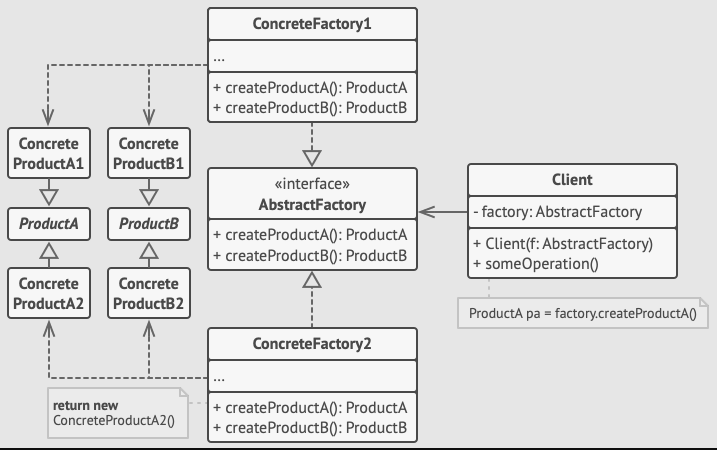


Рисунок 11 – Структура патерна Абстрактна фабрика (Abstract Factory)

1.17 Патерн Будівельник (Builder)

**Будівельник** — це породжувальний патерн проектування, що дає змогу створювати складні об’єкти крок за кроком. Будівельник дає можливість використовувати один і той самий код будівництва для отримання різних відображень об’єктів.

Використовувати патерн Будівельник потрібно, коли:

* процес створення нового об'єкта не повинен залежати від того, з яких частин цей об'єкт складається і як ці частини пов'язані між собою;
* необхідно забезпечити отримання різних варіацій об'єкта у процесі його створення.

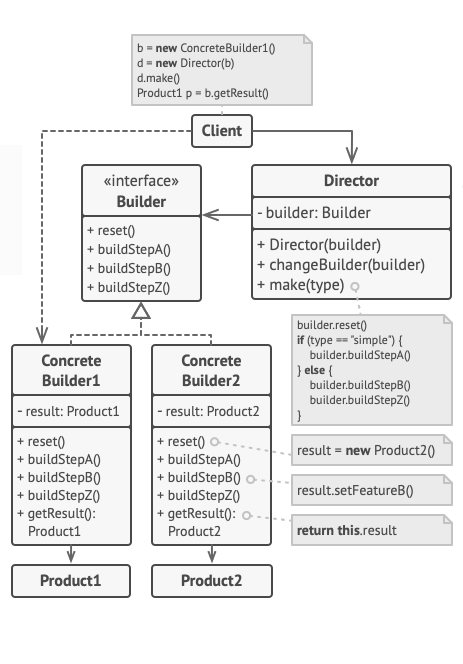


Рисунок 12 – Структура патерна Будівельник (Builder)

1.18 Патерн Прототип (Prototype)

**Прототип** — це породжувальний патерн проектування, що дає змогу копіювати об’єкти, не вдаючись у подробиці їхньої реалізації.

Використовувати патерн Прототип потрібно, коли:

* конкретний тип об'єкта, що створюється, повинен визначатися динамічно під час виконання;
* небажано створення окремої ієрархії класів фабрик для створення об'єктів-продуктів із паралельної ієрархії класів (як це робиться, наприклад, при використанні патерну Абстрактна фабрика);
* клонування об'єкта є кращим варіантом ніж його створення та ініціалізація за допомогою конструктора. Особливо коли відомо, що об'єкт може приймати невелику обмежену кількість можливих станів.

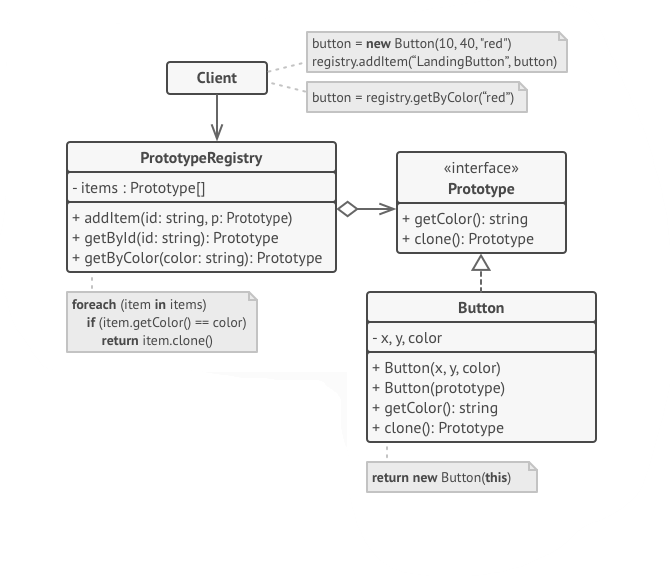


Рисунок 13 – Структура патерна Прототип (Prototype)

1.19 Патерн Одинак (Singleton)

**Одинак** — це породжувальний патерн проектування, який гарантує, що клас має лише один екземпляр, та надає глобальну точку доступу до нього.

Використовувати патерн Одинак потрібно, коли необхідно, щоб для класу існував лише один екземпляр.

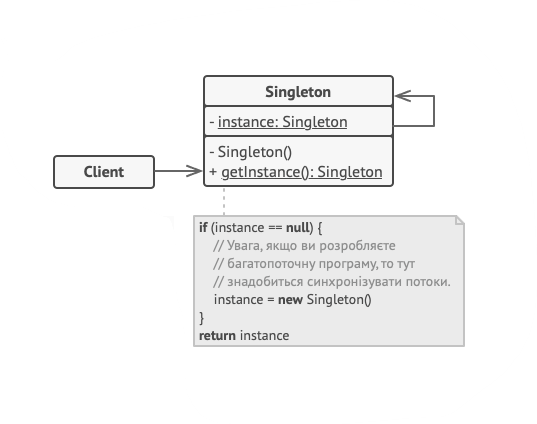


Рисунок 14 – Структура патерна Одинак (Singleton)

1.20 Патерн Адаптер (Adapter)

**Адаптер** — це структурний патерн проектування, що дає змогу об’єктам із несумісними інтерфейсами працювати разом.

Використовувати патерн Адаптер потрібно, коли:

* необхідно використовувати наявний клас, та його інтерфейс відповідає потребам;
* треба використовувати вже існуючий клас спільно з іншими класами, інтерфейси яких не сумісні.

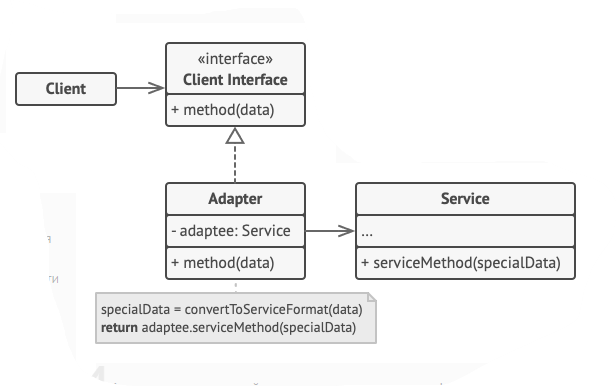


Рисунок 15 – Структура патерна Адаптер (Adapter)

1.21 Патерн Міст (Bridge)

**Міст** — це структурний патерн проектування, який розділяє один або кілька класів на дві окремі ієрархії — абстракцію та реалізацію, дозволяючи змінювати код в одній гілці класів, незалежно від іншої.

Використовувати патерн Міст потрібно, коли:

* треба уникнути постійної прив'язки абстракції до реалізації;
* поруч із реалізацією треба змінювати і абстракцію незалежно друг від друга. Тобто зміни в абстракції не повинні призвести до змін у реалізації.

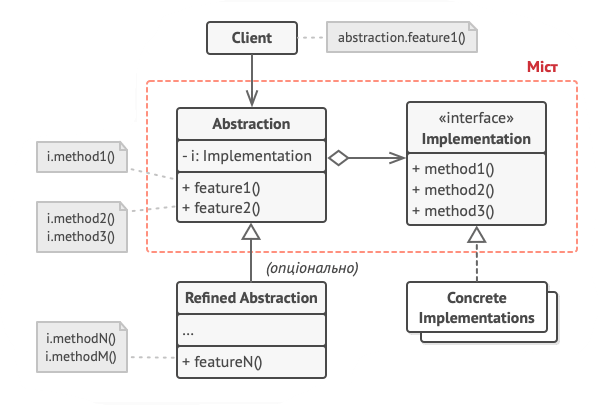


Рисунок 16 – Структура патерна Міст (Bridge)

1.22 Патерн Компонувальник (Composite)

**Компонувальник** — це структурний патерн проектування, що дає змогу згрупувати декілька об’єктів у деревоподібну структуру, а потім працювати з нею так, ніби це одиничний об’єкт.

Використовувати патерн Компонувальник потрібно, коли:

* об'єкти повинні бути реалізовані у вигляді ієрархічної деревоподібної структури;
* клієнти однаково повинні управляти як цілими об'єктами, і їх складовими частинами. Тобто ціле та його частини повинні реалізувати один і той самий інтерфейс.

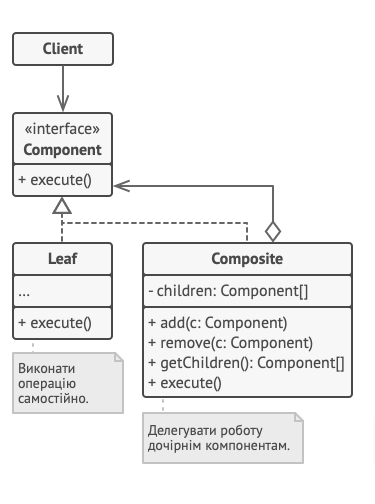


Рисунок 17 – Структура патерна Компонувальник (Composite)

1.23 Патерн Декоратор (Decorator)

**Декоратор** — це структурний патерн проектування, що дає змогу динамічно додавати об’єктам нову функціональність, загортаючи їх у корисні «обгортки».

Використовувати патерн Декоратор потрібно, коли:

* треба динамічно додавати до об'єкта нові можливості. При цьому дані можливості можуть бути зняті з об'єкту;
* застосування наслідування неприйнятне.

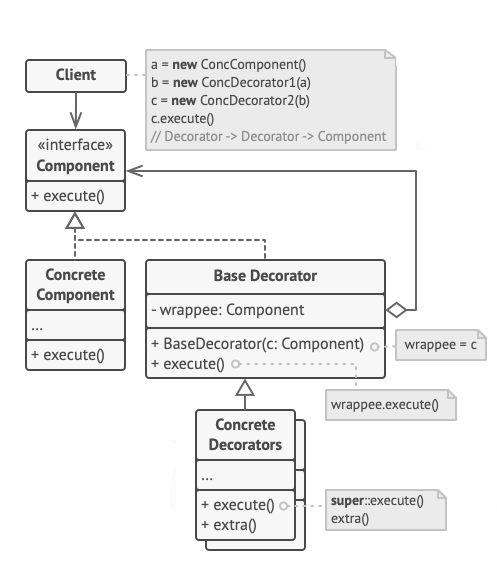


Рисунок 18 – Структура патерна Декоратор (Decorator)

1.24 Патерн Фасад (Facade)

**Фасад** — це структурний патерн проектування, який надає простий інтерфейс до складної системи класів, бібліотеки або фреймворку.

Використовувати патерн Фасад потрібно, коли:

* є складна система, необхідно спростити з нею роботу. Фасад дозволить визначити одну точку взаємодії між клієнтом та системою;
* треба зменшити кількість залежностей між клієнтом та складною системою. Фасадні об'єкти дозволяють відокремити, ізолювати компоненти системи від клієнта та розвивати та працювати з ними незалежно;
* необхідно визначити підсистеми компонентів у складній системі. Створення фасадів для компонентів кожної окремої підсистеми дозволить спростити взаємодію між ними та підвищити їхню незалежність один від одного.

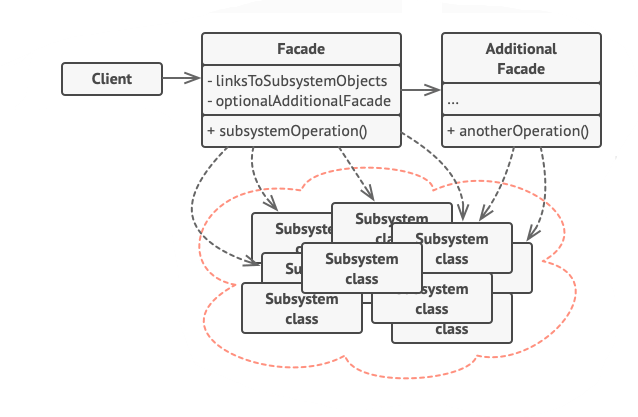


Рисунок 19 – Структура патерна Фасад (Facade)

1.25 Патерн Легковаговик (Flyweight)

**Легковаговик** — це структурний патерн проектування, що дає змогу вмістити більшу кількість об’єктів у відведеній оперативній пам’яті. Легковаговик заощаджує пам’ять, розподіляючи спільний стан об’єктів між собою, замість зберігання однакових даних у кожному об’єкті.

Використовувати патерн Легковаговик потрібно, коли:

* програма використовує велику кількість одноманітних об'єктів, через що відбувається виділення великої кількості пам'яті;
* частину стану об'єкта, що є змінним, можна винести за межі. Винесення зовнішнього стану дозволяє замінити безліч об'єктів невеликою групою загальних об'єктів, що розділяються.

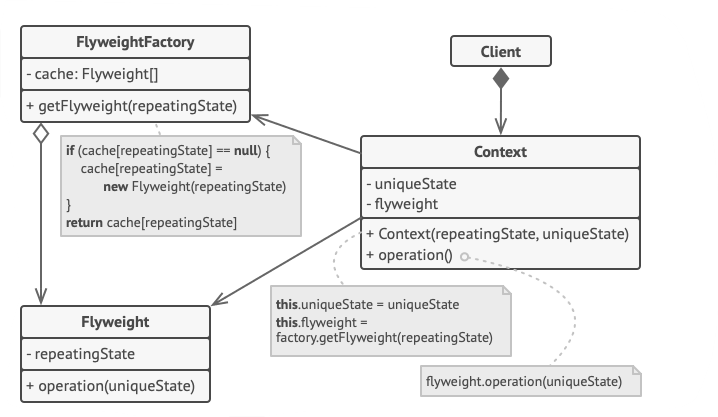


Рисунок 20 – Структура патерна Легковаговик (Flyweight)

2. Докладні відомості про патерни Замісник (Proxy) та Implicit Lock (Скрытая блокировка)

2.1 Патерн Замісник (Proxy)

**Замісник** — це структурний патерн проектування, що дає змогу підставляти замість реальних об’єктів спеціальні об’єкти-замінники. Ці об’єкти перехоплюють виклики до оригінального об’єкта, дозволяючи зробити щось до чи після передачі виклику оригіналові.

Патерн Замісник пропонує створити новий клас-дублер, який має той самий інтерфейс, що й оригінальний службовий об’єкт. При отриманні запиту від клієнта об’єкт-замісник сам би створював примірник службового об’єкта та переадресовував би йому всю реальну роботу.

Завдяки однаковому інтерфейсу об’єкт-замісник можна передати до будь-якого коду, що очікує на сервісний об’єкт.

Використовувати патерн Замісник потрібно, коли:

* треба здійснювати взаємодію через мережу, а об'єкт-проси повинен імітувати поведінки об'єкта у іншому адресному просторі. Використання проксі дозволяє зменшити накладні витрати при передачі даних через мережу. Подібна ситуація ще називається віддаленим заступником (remote proxies);
* потрібно керувати доступом до ресурсу, створення якого потребує великих витрат. Реальний об'єкт створюється тільки тоді, коли він справді може знадобитися, а раніше всі запити щодо нього обробляє проксі-об'єкт. Подібна ситуація ще називається віртуальний заступник (virtual proxies);
* необхідно розмежувати доступ до об'єкта, що викликається, залежно від прав об'єкта, що викликає. Подібна ситуація ще називається заступник, що захищає (protection proxies);
* потрібно вести підрахунок посилань на об'єкт або забезпечити безпечну роботу з реальним об'єктом. Подібна ситуація називається "розумні посилання" (smart reference).

**Приклад із життя**

Платіжна картка — це замісник пачки готівки. І чек, і готівка мають спільний інтерфейс — ними обома можна оплачувати товари. Вигода покупця в тому, що не потрібно носити з собою багато готівки. З іншого боку власник магазину не змушений замовляти клопітку інкасацію коштів з магазину, бо вони потрапляють безпосередньо на його банківський рахунок.

**Реалізація патерна**

1. Визначте інтерфейс, який би зробив замісника та оригінальний об’єкт взаємозамінними.
2. Створіть клас замісника. Він повинен містити посилання на сервісний об’єкт. Частіше за все сервісний об’єкт створюється самим замісником. У рідкісних випадках замісник отримує готовий сервісний об’єкт від клієнта через конструктор.
3. Реалізуйте методи замісника в залежності від його призначення. У більшості випадків, виконавши якусь корисну роботу, методи замісника повинні передати запит сервісному об’єкту.
4. Подумайте про введення фабрики, яка б вирішувала, який з об’єктів створювати: замісника або реальний сервісний об’єкт. Проте, з іншого боку, ця логіка може бути вкладена до створюючого методу самого замісника.
5. Подумайте, чи не реалізувати вам ліниву ініціалізацію сервісного об’єкта при першому зверненні клієнта до методів замісника.

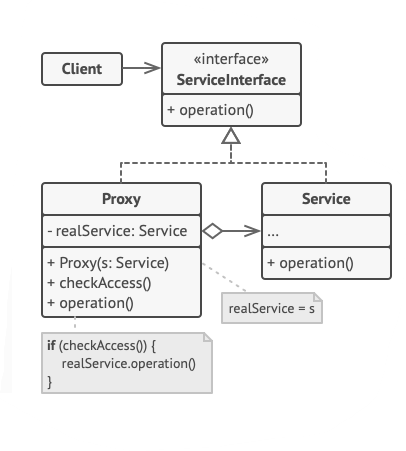


Рисунок 21 – Структура патерна Замісник (Proxy)

## **Переваги**

* Дозволяє контролювати сервісний об’єкт непомітно для клієнта.
* Може працювати, навіть якщо сервісний об’єкт ще не створено.
* Може контролювати життєвий цикл службового об’єкта.

## **Недоліки**

* Ускладнює код програми внаслідок введення додаткових класів.
* Збільшує час отримання відклику від сервісу.

## **Відносини з іншими патернами**

[Адаптер](https://refactoring.guru/uk/design-patterns/adapter) надає класу альтернативний інтерфейс. [Декоратор](https://refactoring.guru/uk/design-patterns/decorator) надає розширений інтерфейс. [Замісник](https://refactoring.guru/uk/design-patterns/proxy) надає той самий інтерфейс.

[Фасад](https://refactoring.guru/uk/design-patterns/facade) схожий на [Замісник](https://refactoring.guru/uk/design-patterns/proxy) тим, що замінює складну підсистему та може сам її ініціалізувати. Але, на відміну від Фасаду, Замісник має такий самий інтерфейс, що і його службовий об’єкт, завдяки чому їх можна взаємозаміняти.

[Декоратор](https://refactoring.guru/uk/design-patterns/decorator) та [Замісник](https://refactoring.guru/uk/design-patterns/proxy) мають схожі структури, але різні призначення. Вони схожі тим, що обидва побудовані на композиції та делегуванні роботи іншому об’єкту. Патерни відрізняються тим, що Замісник сам керує життям сервісного об’єкта, а обгортання Декораторів контролюється клієнтом.

Лістинг 1 – Приклад реалізації патерну на мові С#

using System;

namespace RefactoringGuru.DesignPatterns.Proxy.Conceptual

{

public interface ISubject

{

void Request();

}

class RealSubject : ISubject

{

public void Request()

{

Console.WriteLine("RealSubject: Handling Request.");

}

}

class Proxy : ISubject

{

private RealSubject \_realSubject;

public Proxy(RealSubject realSubject)

{

this.\_realSubject = realSubject;

}

public void Request()

{

if (this.CheckAccess())

{

this.\_realSubject.Request();

this.LogAccess();

}

}

public bool CheckAccess()

{

Console.WriteLine("Proxy: Checking access prior to firing a real request.");

return true;

}

public void LogAccess()

{

Console.WriteLine("Proxy: Logging the time of request.");

}

}

public class Client

{

public void ClientCode(ISubject subject)

{

// ...

subject.Request();

// ...

}

class Program

{

static void Main(string[] args)

{

Client client = new Client();

Console.WriteLine("Client: Executing the client code with a real subject:");

RealSubject realSubject = new RealSubject();

client.ClientCode(realSubject);

Console.WriteLine();

Console.WriteLine("Client: Executing the same client code with a proxy:");

Proxy proxy = new Proxy(realSubject);

client.ClientCode(proxy);

}

}

}

2.2 Патерн Implicit Lock (Скрите блокування)

Дозволяє коду фреймворку або супертипу рівня використовувати локальні блокування.

Суть всіх блокувань у цьому, що з їх використанні не можна робити винятків. Якщо забути написати один рядок коду, який викликає блокування, можна дійти повної марності всієї блокувальної схеми. Не викликавши блокування на читання, навіть якщо інші транзакції його використовують, можна поставити під сумнів актуальність даних. Загалом – якщо щось може бути заблоковане десь, воно має блокуватися скрізь. Ігноруючи це правило, блокувальні стратегії програм ризикують цілісністю даних. Якщо не завершити блокування вчасно, то з даними нічого поганого не трапиться, однак це призведе до провалу продуктивності. Через складність тестування локальних блокувань такі помилки можуть пройти всі тести і не бути виявленими.

Завдання, що використовують блокування, які не можуть бути невиявленими, повинні бути оброблені не для розробника, але приховано для додатка. А той факт, що більшість програм промислового масштабу використовують деякий набір фреймворків, Layer Supertype і генерацію коду, дає великі можливості для впровадження патерну Implicit Lock.

Використовувати патерн Скрите блокування слід застосовувати у всіх додатках (за винятком, мабуть, найпростіших, які не мають інфраструктури). Ризик забути якесь блокування занадто великий, щоб їм можна було знехтувати.

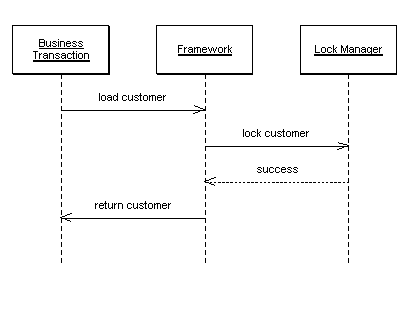


Рисунок 21 – Структура патерна Замісник (Proxy)