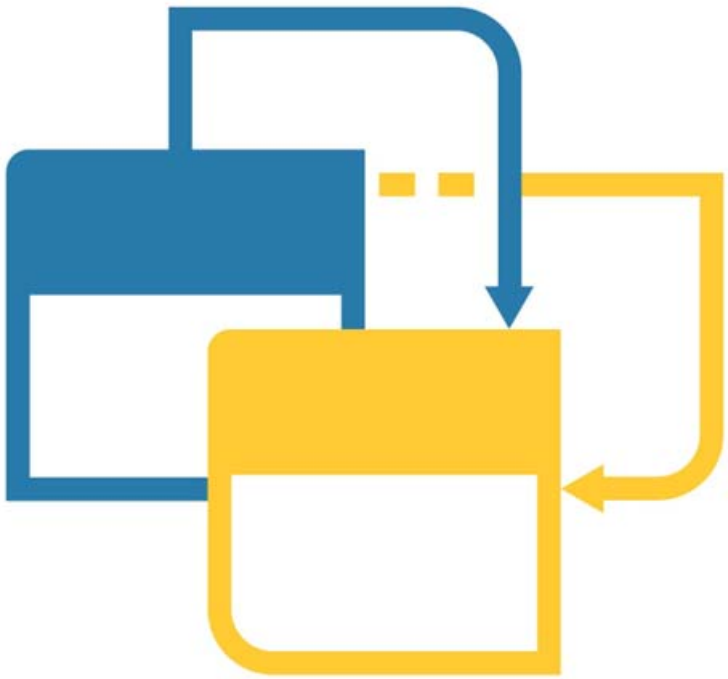
О. В. Войцеховська, О. І. Черняк

ШАБЛОНИ ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ



Міністерство освіти і науки України
Вінницький національний технічний університет

О. В. Войцеховська, О. І. Черняк

ШАБЛОНИ ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

НАВЧАЛЬНИЙ ПОСІБНИК

Вінниця
ВНТУ
2022

УДК 004.415(075.8)
В65

Рекомендовано до видання Вченою Радою Вінницького національного технічного університету Міністерства освіти і науки України (протокол № 6 від 27.01.2022 р.)

Рецензенти:
О. Н. Романюк, доктор технічних наук, професор
С. В. Павлов, доктор технічних наук, професор
О. М. Кузьміна, кандидат технічних наук, доцент

Войцеховська, О. В.

В65 Шаблони проектування програмного забезпечення : навчальний посібник [Електронний ресурс] / О. В. Войцеховська, О. І. Черняк. – Вінниця : ВНТУ, 2022. – (PDF, 100 с.)

В навчальному посібнику розглянуто основні шаблони проектування, які використовуються при розробці програмного забезпечення. Наведено короткий огляд мови UML, що потрібно для вільного сприйняття діаграм, які використовуються при поясненні шаблонів проектування. Крім того, наведено конкретні приклади для кожного патерну з детальним описом класів та інтерфейсів, які в них використовуються, та програмну реалізацію мовою C#. Навчальний посібник розрахований на студентів, що навчаються за навчальним планом підготовки здобувачів вищої освіти за спеціальністю «Комп'ютерна інженерія», а також буде корисним здобувачам інших спеціальностей галузі «Інформаційні технології».

УДК 004.415(075.8)

ЗМІСТ

ВСТУП

Шаблони (або патерни) проектування є парадигмою проблемно-орієнтованого програмування. Вони дають загальне рішення для найбільш відомих задач, що часто виникають при розробці програмного забезпечення. Шаблони проектування використовуються в об'єктно-орієнтованому програмуванні. Вони описують структуру класів або об'єктів, яка є найбільш ефективною при вирішенні поширених проблем. Це дозволяє прискорити процес розробки, використовуючи добре перевірені підходи до побудови програмного коду. Використання шаблонів проектування дозволяє зробити код програми гнучким, зручним для підтримки та більш придатним для повторного використання. Шаблони проектування є загальними для всіх мов програмування, що підтримують ООП. Вони описують ідею, а не конкретну реалізацію.

В цілому шаблони проектування є прикладами ефективного вирішення відомих задач в програмуванні. Дані приклади не призначені для їх копіювання у проекти. В них описано типові проблеми, які можуть виникнути, та запропоновано підходи для усунення цих проблем. Кожен шаблон проектування не є догмою, а може мати різноманітні модифікації й створювати симбіоз з іншими шаблонами. Більш того, багато задач взагалі не потребує використання шаблонів проектування, а деякі можуть використовувати їх лише частково. Тому надмірне або неправильне використання шаблонів проектування лише погіршить код. Однак навіть при написанні програм, що не потребують використання шаблонів проектування, все одно потрібно добре в них орієуватись, щоб передбачити проблеми, які можуть виникнути у майбутньому. Для ефективного використання шаблонів проектування потрібно розуміти їх суть, а також мати ґрунтовні знання і практичний досвід в об'єктно-орієнтованому програмуванні.

Шаблони проектування програмного забезпечення вперше були запропоновані як патерни проектування у відомій книзі «Design patterns. Elements of reusable object-oriented software» авторів E. Gamma, R. Helm, R. Jonson, J. Vlissides. Через вибухову популярність книги їх у технічній літературі жартома називають «бандою чотирьох». Після цього ідея використання шаблонів при проектуванні програмного забезпечення стала стрімко поширюватись, і на сьогодні існує величезна кількість шаблонів різних рівнів для розробки програмного забезпечення у різних технологіях і галузях застосування. У даному посібнику будуть розглянуті саме шаблони проектування «банди чотирьох».

Всі шаблони проектування, що будуть розглянуті, розбито на три групи: породжувальні, структурні і поведінкові.

Породжувальні шаблони описують принципи ефективного створення класів або об'єктів у різних відомих задачах. До них належать такі. як Абстрактна фабрика, Будівельник, Фабричний метод, Прототип і Одинак. Ви-

користання шаблону Абстрактна фабрика дозволяє працювати з різними наборами об’єктів через одну і ту саму групу інтерфейсів. Використання шаблону Будівельник дозволяє створювати об’єкти зі складною структурою, що залежить від різних чинників. Використання шаблону Фабричний метод дозволяє вибирати реалізацію залежно від умови. Використання шаблону Прототип дозволяє швидко отримати копію існуючого об’єкта. Використання шаблону Одинак дозволяє створювати лише один екземпляр класу.

Наведемо випадок використання породжувального шаблону у програмному коді. Наприклад, потрібно створити клас DBConnect для підключення до деякої бази даних, яке може виконуватись у програмі декілька разів. Часто у такому випадку потрібно буде декілька разів створювати екземпляр класу. Це призведе до одночасного існування декількох з’єднань з базою даних, що є, по-перше, дуже затратним у плані ресурсів, і, крім того, додає проблеми у керуванні таким з’єднанням. Для усунення вказаних проблем потрібно створити клас DBConnect за допомогою шаблону Одинак. Тоді екземпляр класу буде створено тільки при першому інстанціюванні, а в усіх інших випадках замість створення нового екземпляра буде надано доступ до вже існуючого. Це дозволить використовувати лише один екземпляр для керування з’єднанням з базою даних.

Структурні шаблони описують принципи ефективного створення складних структур з класів і об’єктів при вирішенні типових задач. До них належать такі, як Адаптер, Міст, Компонувальник, Декоратор, Фасад, Легковаговик, Проксі.

Використання шаблону Адаптер полягає у перетворенні одного інтерфейсу в інший для спільного використання класів з несумісними інтерфейсами. Використання шаблону Міст дозволяє відділяти абстракцію від реалізації за рахунок агрегатування останньої. Це забезпечує їх незалежну модифікацію і розширення. Використання шаблону Компонувальник дозволяє за рахунок рекурсивної композиції створювати ієрархії типів, що забезпечує однакове використання простих і складних об’єктів. Використання шаблону Декоратор дозволяє динамічно додавати об’єкту нові обов’язки за рахунок його композиції в об’єкт-декоратор. Це забезпечує гнучке і вибіркове розширення функціональності окремих об’єктів. Шаблон Фасад дозволяє використовувати один універсальний інтерфейс для роботи зі складною підсистемою, що містить різні, пов’язані між собою компоненти. Це забезпечує зменшення залежності між підсистемами програмного продукту. Використання шаблону Легковаговик дозволяє виділяти загальні властивості класів в окремий тип. Це може забезпечити зменшення витрат ресурсів при створенні великої кількості різнотипних об’єктів. Використання шаблону Проксі дозволяє створювати представника об’єкта. Це забезпечує можливість гнучкого відкладеного створення об’єктів.

Структурні шаблони рівня класів використовують успадкування і реалізацію інтерфейсів. Структурні шаблони рівня об'єктів використовують композицію об'єктів для отримання нових властивостей і функціональності. Наприклад, у шаблоні Проксі створюється об'єкт-замісник, що виконує функцію іншого об'єкта. При цьому інший об'єкт може бути локальним, віддаленим, складним об'єктом, який завантажується за необхідності, або об'єктом, доступ до якого потрібно обмежити. Замісник надає додатковий опосередкований рівень доступу до об'єкта і може обмежувати або розширювати властивості та функціональність останнього.

Поведінкові шаблони описують принципи взаємодії між класами чи об'єктами, що стосуються алгоритмів виконання та розподілення обов'язків. В цих шаблонах увага приділяється не процесу виконання, а зв'язкам між класами та об'єктами. В поведінкових шаблонах рівня класів використовується успадкування класів і реалізація інтерфейсів, а в шаблонах рівня об'єктів використовується композиція об'єктів.

До шаблонів даної групи відносять такі, як Ланцюжок відповідальностей (або Ланцюжок обов'язків), Команда, Інтерпретатор, Ітератор, Медіатор, Знімок, Спостерігач, Стан, Стратегія, Шаблонний метод, Відвідувач.

Використання шаблону Ланцюжок відповідальностей полягає у створенні класів, що агрегатують самі себе, або своїх спадкоємців. Це забезпечує передачу об'єкта по ланцюжку різних обробників доки не зустрінеться потрібний обробник. Використання шаблону Команда дозволяє обробляти команду як об'єкт. Це забезпечує можливість збереження команд, передавання їх у параметри функцій та використання як значення, що повертаються. Використання шаблону Інтерпретатор полягає у поданні різних випадків при вирішенні задачі у вигляді власної простої мови і аналізу їх за допомогою створеного інтерпретатора. Це забезпечує можливість зменшення коду за рахунок кодування випадків, а не створення для кожного випадку окремого класу. Прикладом шаблону Інтерпретатор є регулярні вирази у мові C#. Використання шаблону Ітератор забезпечує послідовний доступ до елементів деякого складного об'єкта, не розкриваючи внутрішньої будови об'єкта. Це дає можливість використання циклічних команд для обходу складових об'єкта. Прикладом шаблону Ітератор є інтерфейс IEnumerable у мові C#. Використання шаблону Медіатор полягає у створенні об'єкта-посередника, в якому інкапсульовані способи взаємодії множини взаємозв'язаних об'єктів. Це дозволяє зменшити кількість зв'язків сильно зв'язаної системи та підвищити її гнучкість за рахунок простої заміни об'єкта-посередника. Використання шаблону Знімок дозволяє довготерміново зберігати поточний внутрішній стан об'єкта для подальшого його відновлення. При цьому сам склад об'єкта не розкривається. Прикладом шаблону Знімок є способи серіалізації у мові C#. Використання шаблону Спостерігач дозволяє встановити залежність декількох об'єктів (спостерігачів) від одного спільного (суб'єкта) таким чином, що при зміні стану суб'єкта надсилаються повідомлення до спостерігачів. Це дає можливість

мати будь-яку кількість спостерігачів, а також незалежно замінювати як суб'єкт, так і спостерігачів. Використання шаблону Стан полягає у створенні об'єкта, що містить посилання деякого базового типу State, до якого можна приєднувати об'єкти станів різних похідних типів. Це дає можливість за допомогою успадкування класів гнучко змінювати поведінку об'єкта так, наче змінився його тип. Використання шаблону Стратегія полягає у створенні в об'єкті посилання базового типу Strategy, до якого можна приєднати об'єкти різних похідних класів, кожен з яких має свою реалізацію деякого алгоритму. Це забезпечує можливість змінювати поведінку об'єкта. Використання шаблону Шаблонний метод полягає у створенні в базовому абстрактному класі шаблонного методу, що містить послідовність викликів інших методів, деякі з яких визначаються у конкретних підкласах. Це забезпечує можливість винести спільну поведінку у бібліотечні класи. Використання шаблону Відвідувач полягає у створенні двох ієрархій класів: одна для елементів, над якими виконуються операції, а друга для відвідувачів, які описують операції над цими елементами. Це відділяє алгоритми обробки від елементів і дозволяє легко додавати і змінювати алгоритми, не змінюючи структуру елементів.

В цілому шаблони проектування являють собою сконцентрований досвід розробки об'єктно-орієнтованих програм з метою повторного використання. Вони описують випробувані практичні методи досягнення розумного балансу між ефективністю і гнучкістю коду при вирішенні низки відомих задач.

Мова програмування C# є мовою високого рівня та має велику і потужну бібліотеку класів, що полегшують створення різноманітного програмного забезпечення. В цій бібліотеці вже реалізовано багато з описаних шаблонів. Тому програмування в C# з застосуванням бібліотеки класів приводить до автоматичного використання шаблонів проектування.

Навчальний посібник містить чотири розділи. У першому розділ дано короткий огляд уніфікованої мови моделювання UML. В другому, третьому та четвертому розділах висвітлено шаблони проектування програмного забезпечення згідно з класифікацією, зокрема, їх призначення, приклади використання та структуру. Приклади програм до кожного шаблону реалізовано мовою C#.

Даний посібник призначено для підготовки здобувачів вищої освіти за спеціальністю «Комп'ютерна інженерія».

1 УНІФІКОВАНА МОВА МОДЕЛЮВАННЯ UML

Уніфікована мова моделювання (англ. Unified Modeling Language) або UML – мова, що використовує графічні позначення для створення абстрактної моделі системи, яка називається UML-моделлю. UML не є мовою програмування. Вона є відкритим стандартом і використовується при створенні об'єктно-орієнтованих моделей для візуалізації дизайну, проектування та документування програмної системи та її функціонування.

Важко відслідкувати відношення та ієрархії в складній програмній системі, яка має тисячі рядків коду. UML поділяє програмну систему на компоненти й підкомпоненти та дозволяє візуально подати архітектуру, дизайн та реалізацію такої програмної системи. Також UML використовується для спілкування розробників програмних продуктів між собою.

UML-модель сумісна з будь-якою мовою програмування, тобто, за нею можна генерувати код такими різними мовами, як C#, C++, Java тощо. Також можна виконати і обернене перетворення, тобто відновити UML-модель за існуючою програмною реалізацією.

При цьому UML має неточну семантику, що іноді може привести до неоднозначного тлумачення та інтерпретації специфікації програм. Також мова містить надлишкові або практично невикористовувані діаграми та конструкції.

UML-модель наводиться у вигляді сутностей й відношень між ними та зображується у вигляді діаграм, які подають систему в такому вигляді, який можна легко транслювати в програмний код.

Сутності – це абстракції, які являють собою об'єкти предметної області та системи.

Відношення з'єднують сутності між собою.

Діаграми групують набори сутностей.

1.1 Сутності в UML

Сутності є основними елементами моделі. Розрізняють чотири **види сутностей** (рис. 1.1):

- структурні – клас, інтерфейс, компонент, активний клас, вузол, кооперація, прецедент;
- поведінкові – взаємодія, стан;
- групувальні – пакети;
- анотаційні – коментарі (примітки).

Структурні сутності – це «іменники» в UML-моделі. Зазвичай це статичні частини моделі, що являють собою або концептуальні, або фізичні елементи.

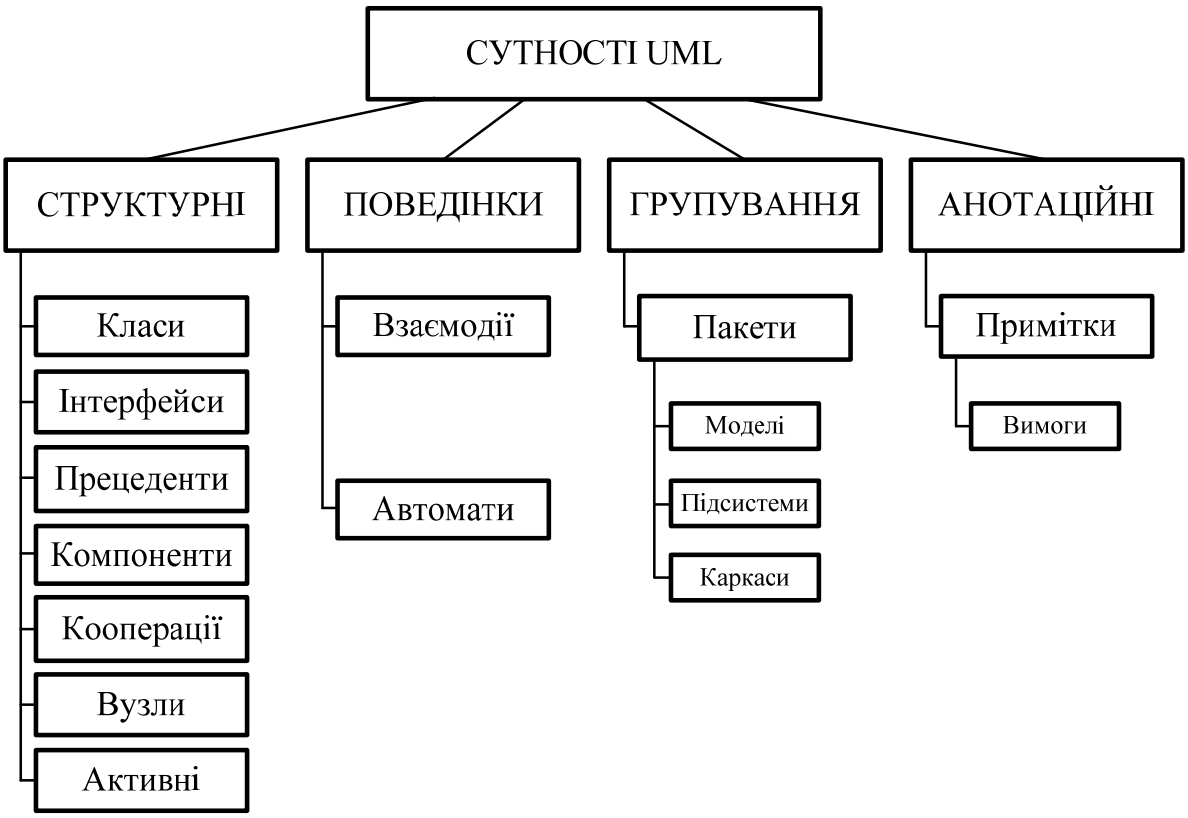


Рисунок 1.1 – Види сутностей в UML

Основними видами структурної сутності в діаграмах класів є *клас* та *інтерфейс*. Графічно *клас* (Class) зображується у вигляді прямокутника, розділеного на три блоки горизонтальними лініями, в яких містяться: ім'я класу, його атрибути (властивості) та операції (методи) класу (рис. 1.2). Блоки атрибутів та операцій можуть бути пустими. Для атрибутів і операцій може бути вказаний один з трьох типів видимості (у вигляді символу зліва в рядку з ім'ям відповідного елемента):

– – private (приватний);
– protected (захищений);
+ – public (відкритий).

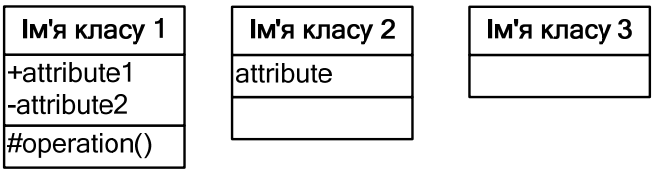


Рисунок 1.2 – Графічне позначення класу в UML

Структурна сутність *інтерфейс* (Interface) графічно зображується подібно до класу, але у інтерфейсу важливими є тільки його методи, тому атрибути відсутні. Над його іменем вказується ключове слово (стереотип) – «interface» (рис. 1.3).

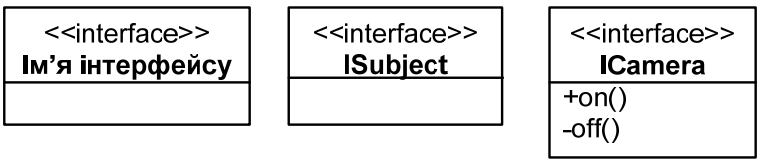


Рисунок 1.3 – Графічне позначення інтерфейсу в UML

Поведінкові сутності – динамічні частини моделей UML. Це «дієслова» моделей, що є поведінкою моделі в часі і просторі.

Сутність *взаємодія* (Interaction) – поведінка, яка полягає в обміні повідомленнями між наборами об'єктів або ролей в певному контексті для досягнення певної мети. Графічно повідомлення зображується у вигляді лінії зі стрілкою, і майже завжди супроводжується ім'ям операції.

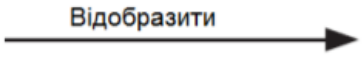


Рисунок 1.4 – Графічне позначення взаємодії

Сутність *стан* (State machine) – це опис різних станів одного компонента протягом життєвого циклу розробки програмного забезпечення. Графічно стан поданий прямокутником із закругленими кутами, зазвичай із вказанням імені й підстанів, якщо такі є (рис. 1.5)

Рисунок 1.5 – Графічне позначення стану

Групувальні сутності – «ящики», по яких можна розкласти модель. Основна сутність – це *пакет*, який використовується для групування семантично пов'язаних елементів моделювання в єдине ціле. Графічно пакет зображується у вигляді теки з закладкою з вказанням імені, а іноді й вмісту (рис. 1.6).

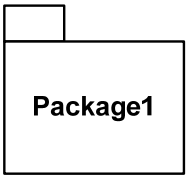


Рисунок 1.6 – Графічне позначення пакета

Анотаційні сутності – це пояснювальні частини UML-моделей, тобто коментарі, які можна застосувати для опису, виділення та пояснення будь-якого елемента моделі. Основна з анотаційних сутностей – *примітка*. Це символ, що слугує для опису обмежень і коментарів, що їх відносять до елемента або набору елементів. Графічно подана прямокутником із загнутим кутом, всередині якого розміщується текстовий або графічний коментар (рис. 1.7).

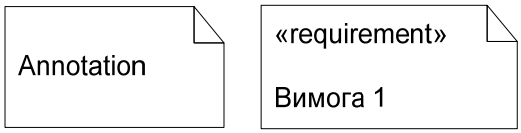


Рисунок 1.7 – Графічне позначення примітки

1.2 Відношення в UML

Відношення показують зв'язки між сутностями. Розрізняють такі **види відношень**.

Залежність (dependency) показує такий зв'язок між двома сутностями, коли зміна незалежної сутності може вплинути на семантику залежної. Графічно залежність зображується пунктирною лінією зі стрілкою, спрямованою від залежної сутності до незалежної (рис. 1.8).

Асоціація (association) – це структурний зв'язок, який показує, що об'єкти однієї сутності пов'язані з об'єктами іншої. Графічно асоціація показується у вигляді суцільної лінії, що з'єднує зв'язувані сутності. Асоціації слугують для здійснення навігації між об'єктами. Якщо потрібна навігація тільки в одному напрямку, він вказується стрілкою на кінці лінії (рис. 1.8).

Окремим випадком асоціації є **агрегування** (aggregation) – відношення виду «ціле»-«частина». Графічно це зображується з допомогою ромба на кінці лінії та розташованого біля цілого.

Ще одним випадком випадком асоціації є **композиція** (composition) – відношення виду «ціле»-«частина», при якому час життя частини збігається з часом життя цілого. Графічно композиція показується з допомогою зафарбованого ромба на кінці лінії та розташованого біля цілого.

Узагальнення (inheritance) – це відношення між сутністю-батьком і сутністю-нащадком, що відображає властивість наслідування для класів і об'єктів. Графічно узагальнення подається у вигляді суцільної лінії, яка закінчується трикутником, спрямованим до сутності-батька (рис. 1.8). Нащадок успадковує структуру (атрибути та поля) і поведінку (методи) батька, але при цьому він може мати нові елементи структури та нові методи.

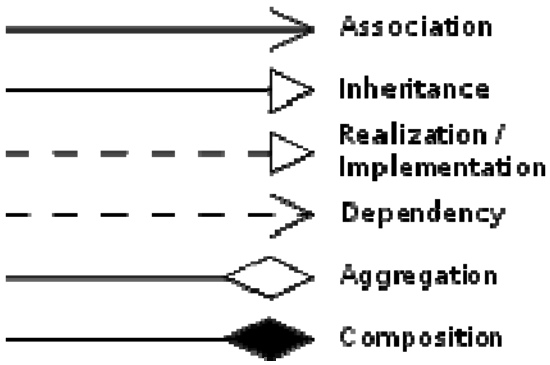


Рисунок 1.8 – Графічне позначення відношень в UML

Реалізація (realization / implementation) – відношення між сутністю, яка визначає специфікацію поведінки (інтерфейс), та сутністю, яка визначає реалізацію цієї поведінки (клас, компонент). Це відношення зазвичай використовується у двох випадках: між інтерфейсами і класами, які реалізують ці інтерфейси, та між варіантами використання, що реалізуються їхніми кооперaціями. Графічно реалізація подається у вигляді пунктирної лінії, яка закінчується трикутником (рис. 1.8).

1.3 Діаграми UML

Діаграма (diagram) − це графічне подання деякої частини графу моделі.

Стандарти UML визначають 13 типів діаграм, які поділені на дві групи.

Структурні діаграми – статичні діаграми, які описують незмінну логічну структуру елементів програмного забезпечення та зображують класи, об’єкти й структури даних, а також відношення, що існують між ними.

Діаграми класів

Діаграми класів є найчастіше використовуваним типом UML-діаграм. Діаграми класів показують статичну структуру системи, зокрема класи, їх атрибути та поведінку, а також відносини між кожним класом. Діаграми класів також можуть відображати обмеження, що накладаються на зв’язки між об’єктами. Використовуються при моделюванні об’єктно-орієнтованих систем.

Діаграми компонентів

Діаграма компонентів розбиває складну систему на більш дрібні компоненти і показує зв’язки між цими компонентами.

Діаграми об’єктів

Діаграми об’єктів показують набір об’єктів і відношень у певний момент виконання системи. Можуть розглядатись як окремий випадок діаграми класів. Це знімок об’єктів системи в певний момент часу. Оскільки

вона показує екземпляри, а не класи, то діаграму об’єктів часто називають діаграмою екземплярів. Використовуються для покращення розуміння структури, показаної на діаграмі класів.

Діаграми розгортання

Діаграми розгортання показують, як програмне забезпечення розгортається на апаратних компонентах (фізичному обладнанні) системи. Ці діаграми корисні для системних інженерів, і вони, зазвичай, показують продуктивність, масштабованість, ремонтопридатність та портативність. Коли апаратні компоненти відображаються один поряд з іншим, легше відстежити всю апаратну сітку та переконатися, що всі елементи враховуються під час розгортання.

Діаграми складеної структури

Ці діаграми являють собою внутрішню структуру класу, тобто показують складові частини складеного об’єкта. Вони також можуть бути використані для подання поведінки співпраці або взаємодії класифікатора з навколишнім середовищем через порти. Вони можуть легко зобразити внутрішні компоненти будь-якого обладнання, щоб більш детально зрозуміти внутрішню роботу.

Діаграми пакетів

Діаграми пакетів використовуються для подання залежностей між різними пакетами в системі. Пакет, зображений у вигляді теки файлу, впорядковує такі елементи моделі, як варіанти використання або класи, у групи. У моделі UML кожен клас може входити лише в один пакет. Пакети можуть входити до складу інших пакетів. Діаграми пакетів зручні у великих за розмірами системах для подання залежностей між основними елементами системи.

Поведінкові діаграми – динамічні діаграми, які показують як змінюються сутності програмного забезпечення під час виконання, як система поводиться та взаємодіє з собою та з користувачами, іншими системами та іншими суб’єктами.

Діаграми діяльності

Діаграми діяльності візуалізують кроки, що виконуються у випадку використання (дії можуть бути послідовними, розгалуженими або одночасними). Цей тип діаграм використовується для подання динамічної поведінки системи, але він також може бути корисним при моделюванні бізнес-процесів. Діаграми діяльності застосовуються, коли потрібно вивчити поведінку кількох об’єктів у кількох прецедентах або потоках.

Діаграми послідовності

Діаграми послідовності показують набір варіантів використання, діючих осіб та їх зв’язків. Діаграми послідовності застосовуються тоді, коли потрібно подивитися на поведінку кількох об’єктів у межах одного прецеденту.

Діаграми взаємодії

Ці діаграми показують потік управління між вузлами, що взаємодіють. Вони містять початкові вузли, кінцеві вузли потоку, кінцеві вузли діяльності, вузли прийняття рішень, вузли злиття та вузли приєднання.

Діаграми комунікації

Діаграми комунікації показують, як об’єкти ставляться один до одного. Вони моделюють спосіб об’єднання та зв’язків за допомогою повідомлень у архітектурному проекті системи. Вони також можуть показувати альтернативні сценарії в межах варіантів використання або операцій, які вимагають співпраці різних об’єктів та взаємодії. Діаграми комунікації допускають довільне розміщення учасників, дозволяють рисувати зв’язки, що показують відносини учасників, та використовувати нумерацію для подання послідовності повідомлень.

Діаграми варіантів використання

Діаграми варіантів використання моделюють як користувачі, зображені у вигляді акторів, взаємодіють із системою.

Діаграми станів

Діаграми станів зображують стани та переходи. Стан належить до різних комбінацій інформації, яку може зберігати об’єкт, і ці діаграми можуть візуалізувати всі можливі стани та способи переходу об’єкта з одного стану в інший. Діаграми станів застосовуються, коли потрібно подивитися на поведінку одного об’єкта в кількох прецедентах. Однак вони не дуже підходять для опису поведінки, що характеризується взаємодією безлічі об’єктів.

Діаграми часу

Діаграми часу є ще однією формою діаграм взаємодії, які показують, як об’єкти взаємодіють між собою в певний період часу. Вони корисні для позначення часових інтервалів між змінами станів різних об'єктів.

Питання для самоперевірки

1. Наведіть приклади структурних сутностей в UML-моделях.
2. Наведіть класифікацію сутностей в UML-моделях.
3. Обгрунтуйте потребу у вивченні UML-діаграм.
4. Охарактеризуйте види відношень в UML-моделях.
5. Охарактеризуйте види UML-діаграм.
6. Поясніть відмінності між діаграмами класів та об’єктів.
7. Поясніть відмінності в графічному позначенні класу та інтерфейсу на UML-діаграмі.
8. Поясніть відмінність між відношеннями композиції та агрегації.
9. Охарактеризуйте специфікатори видимості для членів класу і як їх показати на UML-діаграмі.
10. Охарактеризуйте відношення узагальнення.

2 ПОРОДЖУВАЛЬНІ ШАБЛОНИ ПРОЕКТУВАННЯ

Основним завданням таких патернів є спрощення створення об’єктів. Породжувальні патерни використовують успадкування для різних варіантів створення об’єкта. Вони інкапсулюють знання про те, які конкретні класи використовує система, та приховують, як створюються та об’єднуються екземпляри цих класів.

Абстрактна Фабрика дозволяє використовувати один інтерфейс для створення та обробки наборів об’єктів і мати можливість змінювати типи цих наборів. **Будівельник** дозволяє створювати складні об’єкти. **Фабричний Метод** дозволяє, залежно від умови, вибрати якусь одну реалізацію та її інстанціювати. **Прототип** дозволяє копіювати та тиражувати існуючі об’єкти. **Одинак** дозволяє забезпечити можливість створити лише один екземпляр об’єкта.

При вивченні патернів доцільно ввести такі поняття, як «розробник» і «клієнт». Розробник розробляє певний код відповідно до даного патерну, а клієнт використовує його згідно з цим патерном. Типовим прикладом є створення і використання бібліотек.

2.1 Шаблон Абстрактна Фабрика (Abstract Factory)

Призначення

Патерн Абстрактна фабрика призначений для створення наборів взаємопов’язаних об’єктів різних типів і надання клієнту можливості доступу до цих наборів через загальний інтерфейс. Даний патерн відокремлює створення наборів об’єктів від їх використання. Тому, незалежно від типів створюваних наборів, їх використання буде однаковим і не потребуватиме зміни коду клієнта. Найчастіше патерн застосовується при створенні та використанні бібліотек, які надають різні типи наборів взаємопов’язаних даних. Наприклад, різні типи наборів курсорів або різні типи тем оформлення вікон у Windows тощо.

Переваги

– Спрощує заміну наборів об’єктів.

Недоліки

– Складно додати новий набір, оскільки для цього потрібно створити класи всіх елементів набору і також клас фабрики, у якому потрібно реалізувати функціональність, задану в загальному інтерфейсі.

Коли використовувати даний патерн

– Коли програмне забезпечення не має залежати від того, як створюються, компонуються і подаються його об’єкти.

– Коли потрібно забезпечити спільне використання набору об’єктів.

– Коли програмне забезпечення має конфігуруватись одним з наборів об’єктів.

– Коли бібліотека об’єктів має розкривати лише їх інтерфейси, а не реалізацію.

Загальна структура патерну

– Інтерфейс IAbstractFactory, який містить абстрактну функцію Create(), призначену для створення набору об’єктів.

– Абстрактний клас AbstractElement1OfSet, AbstractElement2OfSet, ... для кожного члена набору. Дані класи є базовими.

– Похідні класи ConcreteElement1OfSet1, ConcreteElement2OfSet1, ..., ConcreteElement1OfSet2, ConcreteElement2OfSet2, ... для кожного члена набору.

– Класи ConcretFactory1, ConcretFactory2, ..., що реалізують інтерфейс IAbstractFactory для кожного набору.

Розробник патерну створює інтерфейс чи абстрактний клас фабрики IAbstractFactory та абстрактні класи набору об’єктів AbstractElement1OfSet, AbstractElement2OfSet, ...

Клієнт створює похідні класи фабрик ConcretFactory1, ConcretFactory2, ... та похідні класи наборів ConcreteElement1OfSet1, ConcreteElement2OfSet1, ..., ConcreteElement1OfSet2, ConcreteElement2OfSet2, ...

Розглянемо приклад задачі, що вирішується створенням програми з використанням патерну Абстрактна фабрика.

Приклад

Умова. Для вирубки лісового масиву управління лісовим господарством сформувало бригаду лісорубів. Відповідно до їх кількості було закуплено інструменти фірми Hunder: бензопили для великих дерев і сокири для невеликих дерев та кущів. Згодом виявилось, що кількість лісорубів потрібно збільшити та, відповідно, закупити додаткові бензопили та сокири. Однак на той час з потрібних інструментів у продажу були лише інструменти фірми Tatra, які й було закуплено.

Завдання. Управління лісовим господарством має мати можливість у циклі роздати бензопили і сокири незалежно від фірми виробника.

Перелік учасників даного прикладу

– Інтерфейс IInstrument Factory – задає функціональність фабрики. Містить абстрактну функцію Create Instruments Set().

– Клас Hunder Factory – реалізує функціональність для фабрики Hunder.

– Клас Tatra Factory реалізує функціональність для фабрики Tatra.

– Абстрактний клас Pyla – є базовим для класів пил. Містить абстрактну функцію Get Name().

– Клас Hunder Pyla – є похідним від класу Pyla.

– Клас Tatra Pyla – є похідним від класу Pyla.

– Абстрактний клас Sokyra – є базовим для класів сокир. Містить абстрактну функцію Get Name().

– Клас Hunder Sokyra – є похідним від класу Sokyra.

– Клас Tatra Sokyra – є похідним від класу Sokyra.

– Клас Program – клієнт, що використовує даний шаблон.

На рис. 2.1 зображено UML-діаграму класів для вирішення задачі з використанням патерну Абстрактна фабрика.

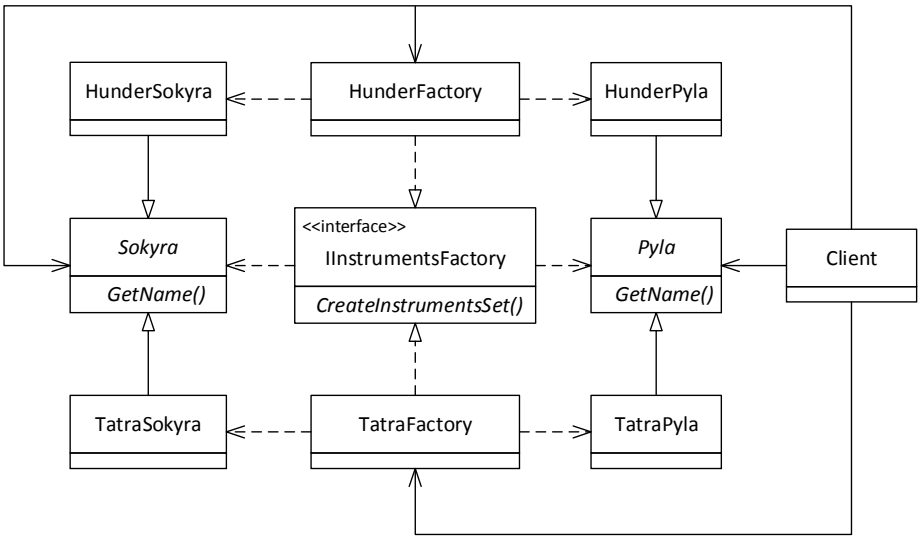


Рисунок 2.1 – UML-діаграма класів шаблону Абстрактна фабрика

Відповідно до зображеної на рисунку діаграми далі наводиться програма вирішення поставленої задачі.

Програмна реалізація

```
using System;
abstract class Pyla
{
 public abstract string GetName();
}
abstract class Sokyra
{
 public abstract string GetName();
}
class HunderPyla : Pyla
{
 public override string GetName() { return "Hunder pyla."; }
}
class HunderSokyra : Sokyra
{
 public override string GetName() { return "Hunder sokyra."; }
```

```
}
class TatraPyla : Pyla
{
 public override string GetName() { return "Tatra pyla."; }
}
class TatraSokyra : Sokyra
{
 public override string GetName() { return "Tatra sokyra."; }
}
interface IInstrumentsFactory
{
 (Pyla, Sokyra) CreateInstrumentsSet() ;
}
class HunderFactory : IInstrumentsFactory
{
 (Pyla, Sokyra) IInstrumentsFactory.CreateInstrumentsSet() { return
(new HunderPyla(), new HunderSokyra()); }
}
class TatraFactory : IInstrumentsFactory
{
 (Pyla, Sokyra) IInstrumentsFactory.CreateInstrumentsSet() { return
(new TatraPyla (), new TatraSokyra()); }
}
class Program
{
 static void DistributeInstruments(IInstrumentsFactory
instrumentsFactory,(Pyla, Sokyra)[] m,int count)
 {
  for (int i = 0; i < count; i++)
   m[i] = instrumentsFactory.CreateInstrumentsSet();
 }
 static void Main(string[] args)
 {
  IInstrumentsFactory instrumentsFactory;
  (Pyla, Sokyra)[] m = new (Pyla, Sokyra)[6];
  instrumentsFactory = new HunderFactory();
  DistributeInstruments(instrumentsFactory, m, 6);
  foreach ((Pyla pyla, Sokyra sokyra) instruments in m)
  {
   Console.WriteLine(instruments.pyla.GetName());
   Console.WriteLine(instruments.sokyra.GetName());
  }
  instrumentsFactory = new TatraFactory();
  DistributeInstruments(instrumentsFactory, m, 6);
  foreach ((Pyla pyla, Sokyra sokyra) instruments in m)
  {
   Console.WriteLine(instruments.pyla.GetName());
   Console.WriteLine(instruments.sokyra.GetName());
  }
 }
}
```

У програмі заміна набору інструментів здійснюється простим присвоєнням інтерфейсу об’єкта іншої фабрики. Для додавання нового набору потрібно, по-перше, створити новий клас для кожного інструмента з нового набору, а по-друге, створити клас нової фабрики, в якому реалізувати функцію CreateInstrumentsSet() інтерфейсу IInstrumentsFactory.

У наданому прикладі ця функція як створюваний набір повертає кортеж.

2.2 Шаблон Будівельник (Builder)

Призначення

Патерн Будівельник відокремлює конструювання складного об’єкта від його подання так, що в результаті одного і того ж процесу конструювання можуть створюватись різні подання.

Переваги

– Дозволяє змінювати внутрішнє подання продукту.

– Ізолює код, який реалізує конструювання та подання.

– Дає більш тонкий контроль над процесом конструювання.

Коли потрібно використовувати даний патерн

– Коли алгоритм створення складного об’єкта не має залежати від того, з яких частин складається об’єкт і як вони стикуються між собою.

– Коли процес конструювання має забезпечувати різні подання об’єктів.

Загальна структура патерну

– Абстрактний клас AbstractBuilder, який містить набір функцій для конструювання складного об’єкта.

– Клас Director, який містить посилання типу Builder і функцію Construct(), що задає послідовність виклику функцій класу Builder.

– Класи ConcreteBuilder1, ConcreteBuilder2, ..., кожен з яких створює відповідний складний об’єкт та реалізує набір функцій для його конструювання і повернення.

Приклад

Умова. В будівельній фірмі є три бригади будівельників, кожна з яких вміє будувати, відповідно, віп-дачі, стандартні дачі і дачі для бідних. Віп-дача має будинок, гараж, басейн і паркан. Стандартна дача має будинок, гараж і паркан. Дача для бідних має будинок, сарай і паркан. Директор фірми приймає замовлення на побудову, хоч не знає, з чого складається кожний вид дач. Однак він знає, в якій послідовності виконується будівництво: спочатку будується будинок, потім гараж, потім басейн, потім сарай, а потім паркан.

Завдання. Клієнт замовляє віп-дачу, стандартну дачу і дачу для бідних, вказуючи для кожної з них бригаду, але не вникаючи в деталі побудови. Директор видає послідовність завдань цій бригаді. Бригада будує дачу і повертає її клієнту.

Список учасників даного прикладу

– Клас Director – містить посилання на абстрактний клас Budivelnik і функцію Construct(), в якій вказано порядок виклику функцій будівельника.

– Абстрактний клас Budivelnik – містить функції побудови дачі. Деякі з них є абстрактними, а деякі мають пусте тіло для того, щоб похідні класи конкретних будівельників могли не знати про них і не реалізовувати.

– Клас BudivelnikVip – похідний від класу Budivelnik. Містить об'єкт класу VipDacha та конкретні реалізації функцій для побудови віп-дачі.

– Клас BudivelnikStd – похідний від класу Budivelnik. Містить об'єкт класу StdDacha та конкретні реалізації функцій для побудови стандартної дачі.

– Клас BudivelnikBidn – похідний від класу Budivelnik. Містить об'єкт класу BidnDacha та конкретні реалізації функцій для побудови бідної дачі.

– Клас VipDacha – містить параметри віп-дачі.

– Клас StdDacha – містить параметри стандартної дачі.

– Клас BidnDacha – містить параметри бідної дачі.

– Клас Program – клієнт, що використовує даний шаблон.

На рисунку 2.2 зображено UML-діаграму класів для вирішення задачі з використанням патерну «Будівельник».

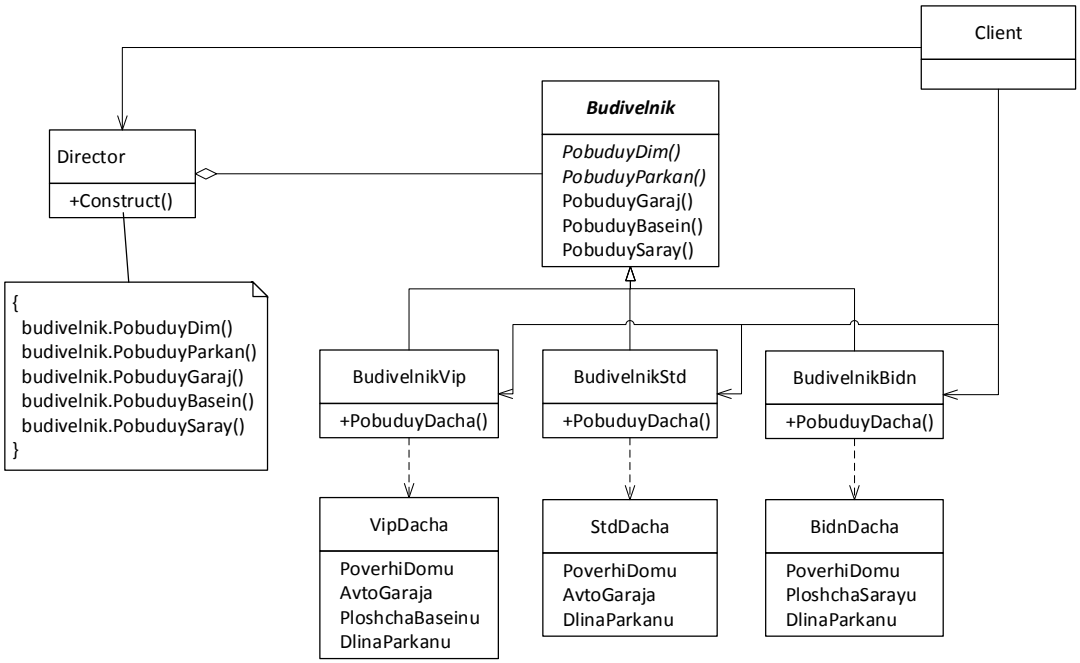


Рисунок 2.2 – UML-діаграма класів прикладу для шаблону Будівельник

Відповідно до зображеної на рисунку діаграми далі наводиться програма вирішення поставленої задачі.

Програмна реалізація

```
using System;
class VipDacha
{
 public int PoverhiDomu { private get; set; }
 public int AvtoGaraga { private get; set; }
 public int DovginaParkanu { private get; set; }
 public int PloshchaBaseinu { private get; set; }
 public override string ToString()
 {
  return "VipDacha: Dim na " + PoverhiDomu + " poverhiv, garaj na "
+ AvtoGaraga + " mashin, parkan " + DovginaParkanu + "m, basein " +
PloshchaBaseinu + "kvm.";
 }
}
class StdDacha
{
 public int PoverhiDomu { private get; set; }
 public int AvtoGaraga { private get; set; }
 public int DovginaParkanu { private get; set; }
 public override string ToString()
 {
  return "StdDacha: Dim na " + PoverhiDomu + " poverhiv, garag na "
+ AvtoGaraga + " mashin, parkan " + DovginaParkanu + "m.";
 }
}
class BidnDacha
{
 public int PoverhiDomu { private get; set; }
 public int DovginaParkanu { private get; set; }
 public int PloschaSarayu { private get; set; }
 public override string ToString()
 {
  return "BidnDacha: Dim na " + PoverhiDomu + " poverhiv, parkan " +
DovginaParkanu + "m, saray " + PloschaSarayu + "kvm.";
 }
}
abstract class Budivelnyk
{
 public abstract void PobuduyDim();
 public abstract void PobuduyParkan();
 public virtual void PobuduyGarag() { }
 public virtual void PobuduyBasein() { }
 public virtual void PobuduySaray() { }
}
class BudivelnykVip : Budivelnyk
```

```
{
 VipDacha vDacha = new VipDacha();
 public override void PobuduyDim() {vDacha.PoverhiDomu=2;}
 public override void PobuduyGarag() {vDacha.AvtoGaraga=2;}
 public override void PobuduyParkan() {vDacha.DovginaParkanu=100; }
 public override void PobuduyBasein() { vDacha.PloshchaBaseinu=40;}
 public VipDacha GetDacha() { return vDacha; }
}
class BudivelnykStd : Budivelnyk
{
 StdDacha sDacha = new StdDacha();
 public override void PobuduyDim() {sDacha.PoverhiDomu = 1;}
 public override void PobuduyGarag() {sDacha.AvtoGaraga = 1;}
 public override void PobuduyParkan() {sDacha.DovginaParkanu = 70;}
 public StdDacha GetDacha() { return sDacha; }
}
class BudivelnykBidn : Budivelnyk
{
 BidnDacha bDacha = new BidnDacha();
 public override void PobuduyDim() {bDacha.PoverhiDomu = 1;}
 public override void PobuduySaray() {bDacha.PloschaSarayu = 40;}
 public override void PobuduyParkan() {bDacha.DovginaParkanu = 50;}
 public BidnDacha GetDacha() { return bDacha; }
}
class Director
{
 public Budivelnyk budivelnyk;
 public Director(Budivelnyk budivelnyk) { this.budivelnyk =
budivelnyk; }
 public void Construct()
 {
  budivelnyk.PobuduyDim();
  budivelnyk.PobuduyGarag();
  budivelnyk.PobuduyBasein();
  budivelnyk.PobuduySaray();
  budivelnyk.PobuduyParkan();
 }
}
class Program
{
 static void Build(Director director, Budivelnyk budivelnik)
 {
  director=new Director(budivelnik);
  director.Construct();
 }
 static void Main(string[] args)
 {
  Director director;
  BudivelnykVip bVip = new BudivelnykVip();
  BudivelnykStd bStd = new BudivelnykStd();
```

```
  BudivelnykBidn bBidn = new BudivelnykBidn();
  director = new Director(bVip);
  director.Construct();
  VipDacha vDacha = bVip.GetDacha();
  Console.WriteLine(vDacha);
  director = new Director(bStd);
  director.Construct();
  StdDacha sDacha = bStd.GetDacha();
  Console.WriteLine(sDacha);
  director = new Director(bBidn);
  director.Construct();
  BidnDacha bDacha = bBidn.GetDacha();
  Console.WriteLine(bDacha);
  Console.ReadKey();
 }
}
```

2.3 Шаблон Фабричний метод (Factory Method)

Призначення

Фабричний метод – це патерн, що визначає інтерфейс для створення об’єктів певного класу, але безпосереднє рішення про те, об’єкт якого класу створювати, визначається у підкласах. Тобто, даний патерн передбачає, що базовий клас делегує створення об’єктів похідним класам.

Коли потрібно використовувати даний патерн

– Якщо наперед невідомо, об’єкти яких типів потрібно створювати.

– Якщо програма має бути незалежною від процесу створення нових об’єктів.

– Якщо потрібно забезпечити можливість легкого введення нових класів для створення об’єктів.

– Якщо потрібно з базового класу делегувати похідним класам створення об’єктів.

Загальна структура патерну

– Абстрактний клас Product – визначає інтерфейс класу, об’єкти якого потрібно створити.

– Класи ConcreteProduct1, ConcreteProduct2, ..., що є похідними від класу Product. Таких класів може бути багато.

– Абстрактний клас Creator – визначає абстрактний фабричний метод FactoryMethod(), що повертає об’єкт типу Product.

– Класи ConcreteCreator1, ConcreteCreator2, ..., що є похідними від класу Creator. Кожен з них визначає свою реалізацію методу FactoryMethod().

Розробник патерну створює абстрактний клас Product і абстрактний клас або інтерфейс Creator, в якому є функція Create(). Клієнт специфікує ці класи.

Переваги

– При розробці, наприклад, бібліотек фабричний метод звільняє проектувальника від необхідності вбудовувати в код створення об’єктів класів користувачів. Він має справу лише з абстрактним класом.

Недоліки

– Клієнт має створювати не лише свій клас ConcreteProduct, але також ще клас ConcreteCreator з реалізацією функції FactoryMethod() навіть тоді, коли потрібно створити лише один об’єкт.

Приклад

Умова. Є класи продуктів NumberedPoint і NamedPoint, об’єкти яких створюються функціями породжувальних класів NumberedPointCreator і NamedPointCreator.

Завдання. Створити у циклі довільну послідовність з об’єктів обох типів.

Перелік учасників даного прикладу

– Абстрактний клас Point – базовий клас для точок різних типів.

– Клас NumerdPoint – похідний клас для нумерованих точок.

– Клас NamedPoint – похідний клас для іменованих точок.

– Інтерфейс IPointCreator – містить функцію FactoryMethod(), яка має викликатись для створення об’єкта типу похідного від класу Point.

– Клас NumeredPointCreator – реалізує інтерфейс IPointCreator для створення нумерованих точок.

– Клас NamedPointCreator – реалізує інтерфейс IPointCreator для створення іменованих точок.

На рисунку 2.3 зображено UML-діаграму класів для вирішення задачі з використанням патерну Фабричний метод.

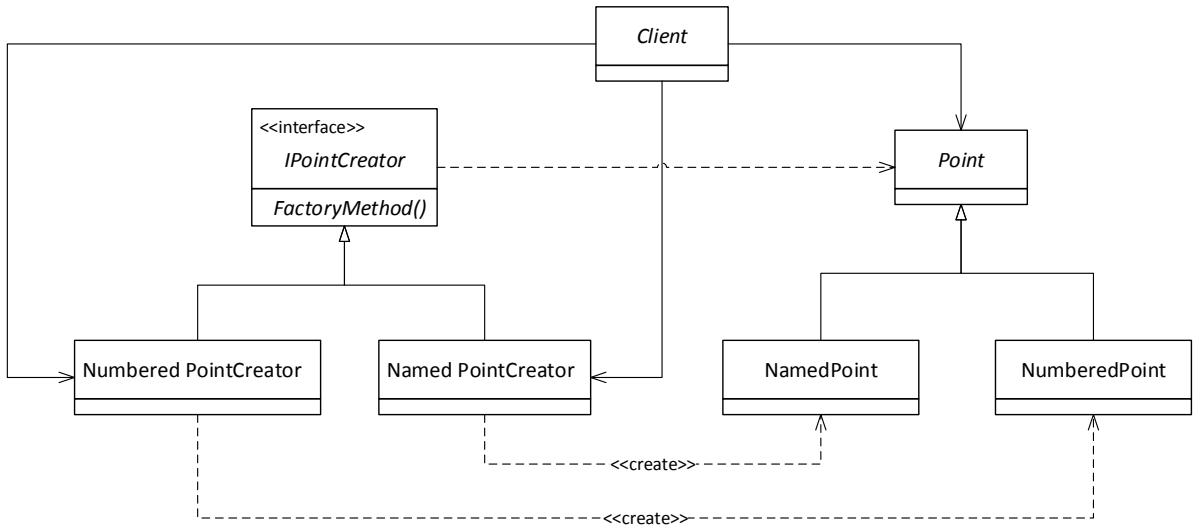


Рисунок 2.3 – UML-діаграма класів для вирішення задачі з використанням патерну Фабричний метод

Відповідно до зображеної на рисунку діаграми далі наводиться програма вирішення поставленої задачі.

Програмна реалізація

```
using System;
namespace FactoryMethod
{
 abstract class Point
 {
  protected int x;
  protected int y;
  public Point(int x, int y) { this.x = x; this.y = y; }
  public abstract void Print();
 }
 class NumberedPoint : Point
 {
  int number;
  public NumberedPoint(int x, int y, int number) : base(x, y)
  {
   this.number = number;
  }
  public override void Print()
  {
   Console.WriteLine($"Numbered point: x={x}, y={y},
number={number}");
  }
 }
 class NamedPoint : Point
 {
  string name;
  public NamedPoint(int x, int y, string name) : base(x, y)
  {
   this.name = name;
  }
  public override void Print()
  {
   Console.WriteLine($"Named point: x={x}, y={y}, name={name}");
  }
 }
 interface IPointCreator
 {
   Point FactoryMethod(int x, int y);
 }
 class NumberedPointCreator : IPointCreator
 {
  static int number = 0;
  public Point FactoryMethod(int x, int y)
  {
   return new NumberedPoint(x, y, ++number);
  }
```

```
 }
 class NamedPointCreator : IPointCreator
 {
  public Point FactoryMethod(int x, int y)
  {
   Console.Write("Input name: ");
   string name = Console.ReadLine();
   return new NamedPoint(x, y, name);
  }
 }
 class Program
 {
  static Point CreatePoint(IPointCreator creator, int x, int y)
  {
   return creator.FactoryMethod(x, y);
  }
  static void Main(string[] args)
  {
   Point[] points =
   {
    CreatePoint(new NamedPointCreator(), 10, 20),
    CreatePoint(new NumberedPointCreator(),50,80),
    CreatePoint(new NumberedPointCreator(),130,40),
    CreatePoint(new NamedPointCreator(), 90, 330)
   };
   foreach (Point point in points)
    point.Print();
  }
 }
}
```

2.4 Шаблон Прототип (Prototype)

Призначення

Прототип (Prototype) – це породжувальний патерн, який дозволяє створювати новий екземпляр класу на основі вже наявного об’єкта. При цьому всі значення нового об’єкта мають бути ідентичні значенням початкового. Тобто, даний патерн визначає метод клонування об’єкта.

Коли потрібно використовувати даний патерн

– Коли конкретний тип створюваного об’єкта має визначатися динамічно під час виконання.

– Коли небажано створення окремої ієрархії класів фабрик для створення об’єктів-продуктів з паралельної ієрархії класів (як це робиться, наприклад, при використанні патерну Абстрактна фабрика).

– Коли клонування об’єкта є кращим варіантом, ніж його створення та ініціалізація за допомогою конструктора. Особливо, коли відомо, що об’єкт може приймати невелике обмежене число можливих станів.

Загальна структура патерну

– Інтерфейс ICloneable – містить функцію Clone(), яка повертає тип object, призначену для створення та повернення копії даного об’єкта.

– Клас ConcreteType – реалізує інтерфейс ICloneable. У тілі функції Clone() створюється посилання типу ConcreteType. Якщо у класі є змінні-значення, то копія створюється шляхом виклику функції MemberwiseClone() об’єкта-джерела. Інакше викликається оператор new. Якщо у класі є змінні-посилання, то вони вручну копіюються з об’єкта в копію. Після чого функція Clone() повертає копію.

– Клієнт – створює об’єкт класу ConcreteType і задає його змінні. Потім створює посилання типу ConcreteType, якому присвоює результат функції Clone() об’єкта, приведений до типу ConcreteType.

На рисунку 2.4 зображена загальна UML-діаграма класів даного патерну.

Приклад

Умова. Є клас Vidrizok, який містить змінні A і B класу Tochka, а також довжину типу double. Клас Tochka містить координати x та y. У конструкторі вводяться координати його кінців і обчислюється довжина.

Завдання. Створити два паралельних відрізки однакової довжини.

Мотивація використання патерну. Для створення відрізка CD паралельного і рівного за довжиною відрізку AB виклик конструктора приводить до виконання значної кількості команд. Більш простим рішенням буде копіювання AB у CD і зсув координат його кінців на deltaX і deltaY відповідно.

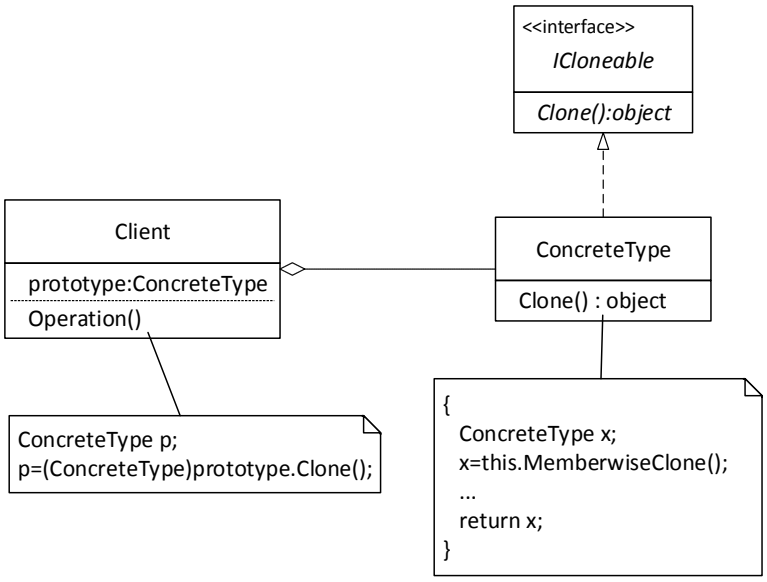


Рисунок 2.4 – Загальна UML-діаграма класів патерну Прототип

Перелік учасників даного прикладу

– Інтерфейс ICloneable – містить функцію Clone(), яка повертає тип object і призначена для створення і повернення копії даного об’єкта.

– Клас Tochka – реалізує інтерфейс ICloneable. Містить дві змінні-значення x та y цілого типу. Має конструктор з параметрами, функцію Clone() та функцію Zsuv() для зміни координат.

– Клас Vidrizok – реалізує інтерфейс ICloneable. Містить дві змінні-посилання A та B типу Tochka і одну змінну-значення dlina типу double. Має конструктор з параметрами, функцію Clone() та функцію Zsuv() для зміни координат кінців. Його клонування виконується шляхом спочатку поверхневого, а потім глибинного копіювання.

– Client – Клас Program, що створює змінну і посилання типу Vidrizok. Виконує клонування змінної у посилання.

На рисунку 2.5 зображена UML-діаграма класів для даного прикладу.

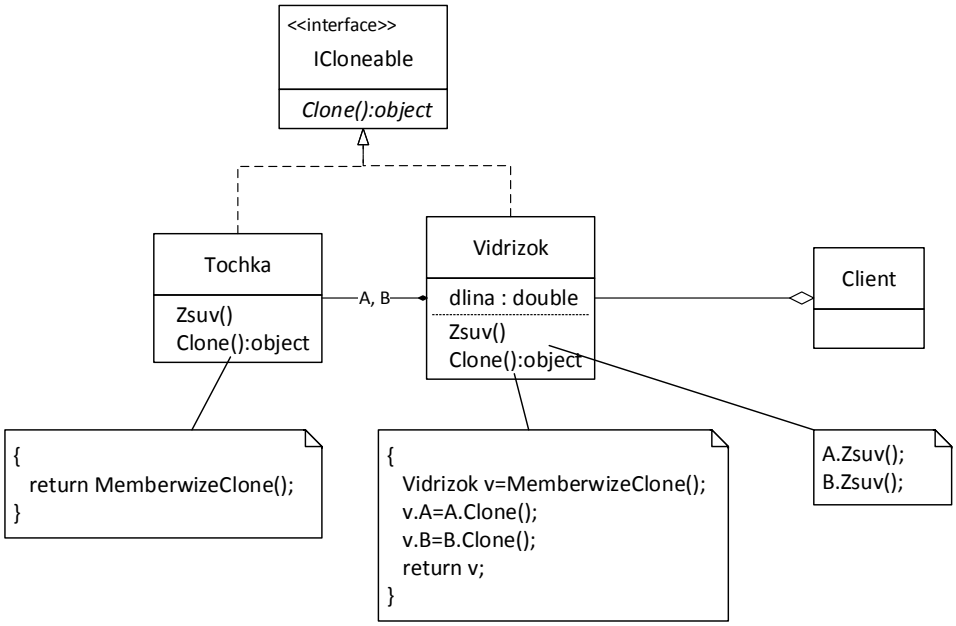


Рисунок 2.5 – UML-діаграма класів для вирішення задачі
з використанням патерну Прототип

Відповідно до зображеної на рисунку діаграми далі наводиться програма вирішення поставленої задачі.

Програмна реалізація

```
using System;
using static System.Math;
using static System.Console;
namespace Prototype
{
 class Tochka:ICloneable
```

```
 {
  public int x { get; private set; }
  public int y { get; private set; }
  public Tochka(int x,int y) { this.x = x; this.y = y; }
  public object Clone() { return this.MemberwiseClone(); }
  public void Zsuv(int deltaX,int deltaY) { x += deltaX; y += deltaY; }
  public override string ToString() { return $"x={x}, y={y}"; }
 }
 class Vidrizok:ICloneable
 {
  Tochka A;
  Tochka B;
  double dlina;
  public Vidrizok()
  {
   int x, y;
   Write("Vvedit A.x: ");
   x = int.Parse(ReadLine());
   Write("Vvedit A.y: ");
   y = int.Parse(ReadLine());
   A = new Tochka(x, y);
   Write("Vvedit B.x: ");
   x = int.Parse(ReadLine());
   Write("Vvedit B.y: ");
   y = int.Parse(ReadLine());
   B = new Tochka(x,y);
   dlina = Sqrt(Pow(A.x - B.x, 2) + Pow(A.y - B.y, 2));
  }
  public object Clone()
  {
   Vidrizok v = (Vidrizok)this.MemberwiseClone();
   v.A = (Tochka)A.Clone();
   v.B = (Tochka)B.Clone();
   return v;
  }
  public void Zsuv(int deltaX, int deltaY)
  {
   A.Zsuv(deltaX, deltaY);
   B.Zsuv(deltaX, deltaY);
  }
  public override string ToString()
  {
   return $"Vidrizok: A:({A}); B: ({B}). Dlina={dlina}";
  }
 }
 class Program
 {
  static void Main(string[] args)
```

```
  {
   Vidrizok AB = new Vidrizok();
   Vidrizok CD = (Vidrizok)AB.Clone();
   int deltaX, deltaY;
   Write("Vvedit deltaX: ");
   deltaX = int.Parse(ReadLine());
   Write("Vvedit deltaY: ");
   deltaY = int.Parse(ReadLine());
   CD.Zsuv(deltaX, deltaY);
   WriteLine(AB);
   WriteLine(CD);
  }
 }
}
```

2.5 Шаблон Одинак (Singleton)

Призначення

Одинак є породжувальним шаблоном, який забезпечує створення лише одного об’єкта для певного класу, а також надає точку доступу до цього об’єкта.

Коли потрібно використовувати даний патерн

Коли потрібно, щоб для класу існував лише один екземпляр.

Переваги

– Контрольований доступ до єдиного екземпляра.

– Підтримка поліморфізму. Можна створювати похідні від Singleton класи.

– Можна дозволити не один, а певне задане число екземплярів.

– Якщо екземпляр Одинака не використовується, то він і не створюється.

Недоліки

– Проблеми з коректним знищенням екземпляра.

– Похідні від Одинака класи не є за замовчуванням Одинаками. Потрібні додаткові дії.

– При кожному звертанні до Одинака відбувається перевірка умови, яка у більшості випадків є зайвою.

У найпростішому варіанті шаблон Одинак містить один клас (наприклад, Singleton), який має закриту статичну змінну типу Singleton (наприклад, private static Singleton instance), захищений конструктор без параметрів (protected Singleton()), статичний конструктор (static Singleton()) і відкриту статичну функцію (наприклад, public Singleton GetInstance()) або відкриту властивість з аксесором get (напиклад, public Singleton Instance). Змінна instance є посиланням на унікальний екземпляр Одинака. Захище-

ний конструктор має доступ protected тому, що це не дозволяє створювати екземпляри за допомогою оператора new, але при цьому дозволяє успадковувати Одинака. Функція GetInstance() повертає посилання на унікальний екземпляр Одинака, створюючи його за необхідності.

На рисунку 2.6 зображена загальна UML-діаграма класів для даного патерну.

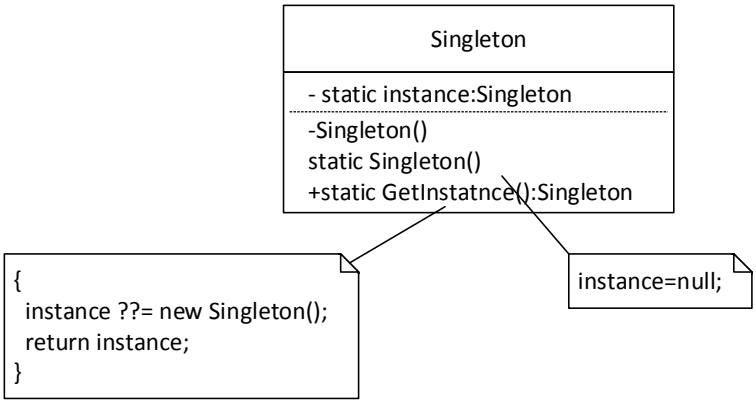


Рисунок 2.6 – UML-діаграма класів патерну Одинак

Приклад

За допомогою патерну Singleton забезпечити унікальність екземпляра класу CenterPoint, який має координати x та y.

Програмна реалізація

```
using System;
namespace Singleton
{
 class CenterPoint
 {
  int x;
  int y;
  static CenterPoint center;
  public static CenterPoint Center
  {
   get
   {
    center ??= new CenterPoint();
    return center;
   }
  }
  protected CenterPoint() { x = 0; y = 0; }
  static CenterPoint() { center = null; }
  public void Set(int x, int y) { this.x = x; this.y = y; }
  public override string ToString()
  {
```

```
    return "x=" + x + ", y=" + y;
   }
  }
  class Program
  {
   static void Main(string[] args)
   {
    CenterPoint center1 = CenterPoint.Center;
    CenterPoint center2 = CenterPoint.Center;
    center1.Set(10, 20);
    Console.WriteLine(center1);
    Console.WriteLine(center2);
    center2.Set(30, 40);
    Console.WriteLine(center1);
    Console.WriteLine(center2);
   }
  }
}
```

Питання для самоперевірки

1. Коротко охарактеризуйте породжувальні шаблони проектування.
2. Наведіть переваги та недоліки використання шаблону Прототип.
3. Наведіть UML-діаграму для шаблону Фабричний метод.
4. Коли доцільно використовувати шаблон Абстрактна фабрика?
5. Охарактеризуйте шаблон Одинак. Наведіть приклад, пов’язаний з реальним життям.
6. Охарактеризуйте які можливості дає використання шаблону Будівельник. Наведіть приклад з реального життя.
7. Поясніть в яких випадках використовується патерн Фабричний метод. Наведіть його переваги та недоліки.
8. Наведіть спосіб забезпечення контролю над створенням екземплярів класу в шаблоні Одинак.
9. Наведіть UML-діаграму для шаблону Будівельник.
10. Охарактеризуйте шаблон Одинак.
11. Поясніть в яких випадках використовується патерн Будівельник. Наведіть його переваги та недоліки.
12. Наведіть UML-діаграму для шаблону Одинак.

3 СТРУКТУРНІ ШАБЛОНИ ПРОЕКТУВАННЯ

Ці патерни проектування стосуються композиції класів та об’єктів. Концепція успадкування використовується для складання інтерфейсів та визначення способів складання об’єктів для отримання нових функціональностей.

Міст відділяє абстаракцію від її реалізації так, що вони можуть змінюватись незалежно. **Адаптер** дозволяє класам працювати разом, що було неможливе через несумісні інтерфейси. **Композит** дозволяє клієнтам однаково обробляти окремі об’єкти та композиції об’єктів. **Проксі** надає замінник для іншого об’єкта для контролю доступу до нього. **Декоратор** забезпечує гнучку альтернативу підкласу для розширення функціональних можливостей. **Фасад** забезпечує єдиний інтерфейс для набору інтерфейсів у підсистемі.

3.1 Шаблон Міст (Bridge)

Призначення

Міст (Bridge) – це структурний шаблон проектування, який розділяє декілька класів на дві окремі ієрархії – абстракцію та її реалізацію – та дозволяє змінювати одну гілку класів незалежно від іншої. Зміна класу реалізації не вимагає перекомпіляції класу абстракції та його клієнтів.

Коли потрібно використовувати даний патерн

– Коли потрібно розділити монолітний клас, який містить кілька різних реалізацій певної функціональності.

– Коли клас потрібно розширювати в двох незалежних площинах.

– Коли потрібно мати можливість змінювати реалізацію під час виконання програми.

Переваги

– Приховує особливості та проблеми реалізації від об’єкта-клієнта.

– Відділення реалізацій від об’єктів, які їх використовують, підвищує розширюваність.

Недоліки

– Введення додаткових класів ускладнює код.

Загальна структура патерну

– Клас Abstraction – визначає інтерфейс для об’єктів, що реалізуються, та зберігає посилання на об’єкт типу Implementor.

– Клас RefinedAbstraction – розширює інтерфейс, визначений класом Abstraction.

– Клас Implementor – визначає загальний інтерфейс для конкретних класів реалізації.

– Клас ConcreteImplementor – реалізує інтерфейс Implementor та визначає його конкретну реалізацію. Цей інтерфейс не має точно відповідати інтерфейсу Abstraction. Зазвичай інтерфейс реалізації забезпечує лише примітивні операції, а Abstraction визначає операції вищого рівня на основі цих примітивів. Тобто класи, отримані від Abstraction, використовують класи, отримані від Implementor, не знаючи, який саме ConcreteImplementor використовується.

Клієнт має знати лише про абстракцію, а реалізацію можна приховати від нього.

На рисунку 3.1 зображено загальну UML-діаграму класів для даного патерну.

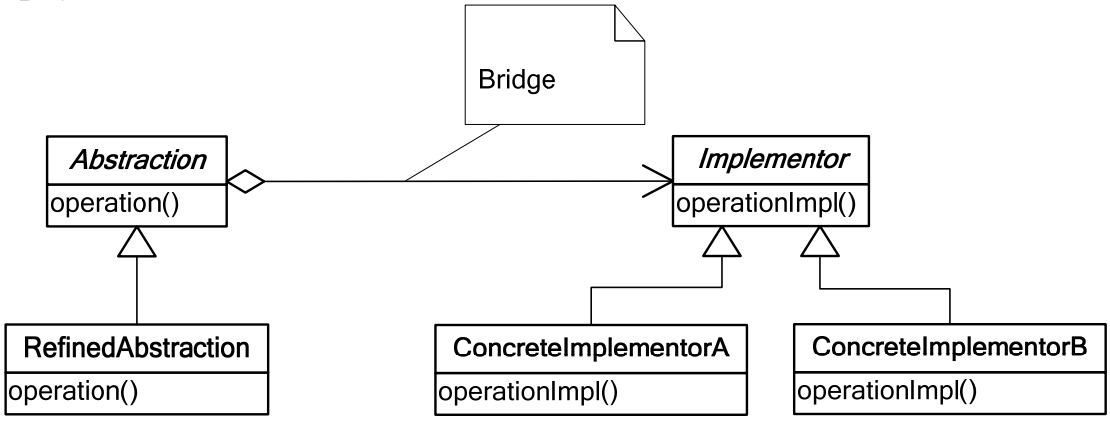


Рисунок 3.1 – Загальна UML-діаграма класів патерну Міст

Приклад

Умова. Задано клас геометричних фігур Figure та два його нащадки – квадрат Rectangle та круг Circle.

Завдання. Розширити ієрархію фігур за кольором, наприклад, отримати червоні та сині фігури.

Мотивація використання патерну. Найпростішим рішенням є використання успадкування, а саме створення чотирьох класів-нащадків: RedCircle, BlueCircle, RedRectangle, BlueRectangle. Однак при такому підході при зростанні кількості потрібних кольорів (або фігур) стрімко зростає кількість класів-нащадків, тобто розширення сильно ускладнюється та втрачається гнучкість коду. Краще створити окремий клас кольору (IColor) з класами-нащадками Red та Blue. При цьому клас Figure отримає посилання на об’єкт IColor і буде, за потреби, делегувати йому роботу. Це і буде мостом між Figure та IColor, тобто, при додаванні нових класів кольорів не потрібно буде звертатись до класів фігур і навпаки.

Перелік учасників даного прикладу

– Інтерфейс IColor – містить функцію GetColor().

– Класи Red та Blue – реалізують інтерфейс IColor. Містять функцію GetColor(), яка повертає колір відповідно до назви класу.

– Клас Figure – містить властивість Color типу IColor, конструктор з параметром та абстрактний метод Draw() для виведення результату.

– Класи Rectangle та Circle наслідують клас Figure, в яких переозначено метод Draw() та міститься конструктор, що звертається до конструктора базового класу.

– Client – Клас Program, що створює дві змінних типу Figure та викликає для них метод Draw().

На рисунку 3.2 зображено UML-діаграму класів для даного прикладу.

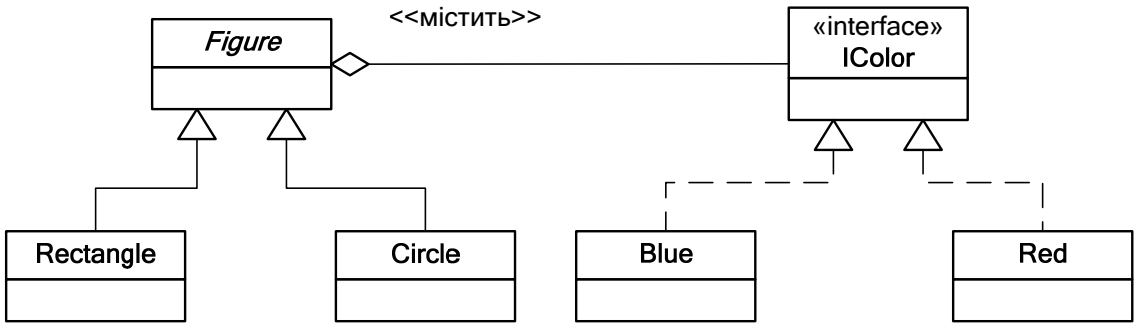


Рисунок 3.2 – UML-діаграма класів для вирішення задачі з використанням патерну Міст

Відповідно до зображеної на рисунку діаграми далі наводиться код програми для вирішення поставленої задачі.

Програмна реалізація

```
using System;
namespace Bridge
{
 class Program
 {
  public static void Main(string[] args)
  {
   Figure rectangle = new Rectangle(new Red());
   Figure circle = new Circle(new Blue());
   rectangle.Draw();
   circle.Draw();
  }
 }
 interface IColor
 {
  string GetColor();
 }
 class Red : IColor
 {
  public string GetColor()
  {
```

```
   return "red";
  }
 }
 class Blue : IColor
 {
  public string GetColor()
  {
   return "blue";
  }
 }
 abstract class Figure
 {
  protected IColor Color { get; }
  protected Figure(IColor color)
  {
    Color = color;
  }
  public abstract void Draw();
 }
 class Rectangle : Figure
 {
  public Rectangle(IColor color) : base(color) { }
  public override void Draw()
  {
   Console.WriteLine($"Draw a {Color.GetColor()} rectangle");
  }
 }
 class Circle : Figure
 {
  public Circle(IColor color) : base(color) { }
  public override void Draw()
  {
   Console.WriteLine($"Draw a {Color.GetColor()} circle");
  }
 }
}
```

3.2 Шаблон Адаптер (Adapter)

Призначення

Адаптер (Adapter) – це структурний шаблон проектування, який дає можливість об’єктам із несумісними інтерфейсами працювати разом. Він адаптує функціональність несумісного з системою об’єкта через інший, відомий цій системі, інтерфейс.

Коли потрібно використовувати патерн

– Коли потрібно використати клас, інтерфейс якого не відповідає решті коду програми.

– Коли потрібно використати кілька існуючих підкласів, але в них немає спільної функціональності, а розширити основний клас неможливо.

Переваги

– Відокремлює та приховує від клієнта деталі перетворення різних інтерфейсів.

Недоліки

– Ускладнює код програми через введення додаткових класів.

Загальна структура патерну

Клас Adaptee – це клас, який клієнт не може використовувати безпосередньо, оскільки Adaptee має незрозумілий йому інтерфейс.

– Клас Target описує алгоритм, через який клієнт може працювати з іншими класами.

– Клас Adapter – це клас, який може одночасно працювати і з класом Target, і з класом Adaptee. Він реалізує клієнтський інтерфейс і містить посилання на об’єкт сервісу.

– Client – це клас, який містить існуючу бізнес-логіку програми.

На рисунку 3.3 зображено UML-діаграму класів для даного патерну.

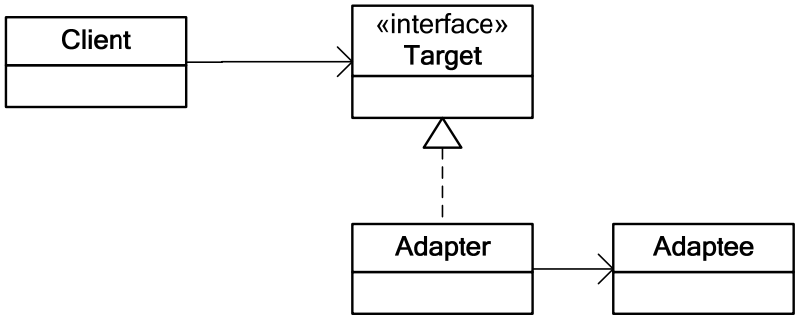


Рисунок 3.3 – UML-діаграма класів патерну Адаптер

Приклад

Умова. Працює e-commerce веб-сайт, який дає можливість користувачам робити покупки та оплату в Інтернеті. Сайт інтегрований з постачальником платіжних послуг XPay, за допомогою якого користувачі можуть оплачувати рахунки своєю кредитною карткою.

Завдання. Змінити постачальника платіжних послуг на YPay, інтерфейс якого несумісний з інтерфейсом Xpay, і впровадити в код.

Мотивація. Немає можливості змінити інтерфейс нового постачальника платіжних послуг Ypay. Тому потрібно використати інтерфейс попереднього постачальника платіжних послуг і адаптувати його для інтерфейсу нового постачальника послуг.

Перелік учасників даного прикладу

– Інтерфейс Xpay – містить набір сетерів і метод геттера, який використовується для отримання інформації про кредитну картку та ім’я клієнта.

– Клас XpayImpl – визначає реалізацію інтерфейсу Xpay.

– Інтерфейс Ypay – визначає інтерфейс нового постачальника послуг, містить набір різних властивостей, які потрібно реалізувати в коді. Однак потрібен адаптер, з допомогою якого можна виконати вимоги постачальника, щоб обробити платіж, а також внести мінімум змін до існуючого коду.

– Клас XpayToYpayAdapter – це адаптер, який реалізує інтерфейс Ypay, оскільки він має імітувати об’єкт типу Ypay. Об’єкт передається в адаптер через його конструктор. Метод setProp() використовується для встановлення властивостей Xpay в об’єкт Ypay.

– Client – Клас Program, який створює об’єкт Xpay і встановлює його властивості. Потім створює адаптер, передає в його конструктор цей об’єкт Xpay і призначає його інтерфейсу Ypay.

Програмна реалізація

```
using System;
using System.Collections.’Generic;
using System.Linq;
using System.Text;
namespace Adapter
{
 class Program
 {
  public static void Main(string[] args)
  {
   // Object for Xpay
   Xpay xpay = new XpayImpl();
   xpay.CreditCardNo = "4489565874102365";
   xpay.CustomerName = "Olena V";
   xpay.CardExpMonth = "09";
   xpay.CardExpYear = "25";
   xpay.CardCVVNo = (short)235;
   xpay.Amount = 2565.23;
   Ypay ypay = new XpayToYpayAdapter(xpay);
   testPayD(ypay);
   Console.ReadLine ( );
  }
  private static void testPayD(Ypay payN)
  {
    Console.WriteLine(payN.CardOwnerName);
    Console.WriteLine(payN.CustCardNo);
    Console.WriteLine(payN.CardExpMonthDate);
    Console.WriteLine(payN.CVVNo);
    Console.WriteLine(payN.TotalAmount);
  }
 }
 public interface Xpay
 {
  string CreditCardNo { get; set; }
```

```
 string CustomerName { get; set; }
 string CardExpMonth { get; set; }
 string CardExpYear { get; set; }
 short CardCVVNo { get; set; }
 double Amount { get; set; }
}
public class XpayImpl : Xpay
{
 private string creditCardNo;
 private string customerName;
 private string cardExpMonth;
 private string cardExpYear;
 private short cardCVVNo;
 private double amount;
 public string CreditCardNo
 {
  get {return creditCardNo;}
  set { creditCardNo = value;}
 }
 public string CustomerName
 {
  get {return customerName;}
  set {customerName = value;}
 }
 public string CardExpMonth
 {
  get {return cardExpMonth;}
  set { cardExpMonth = value;}
 }
 public string CardExpYear
 {
  get {return cardExpYear;}
  set { cardExpYear = value;}
 }
 public short CardCVVNo
 {
  get {return cardCVVNo;}
  set { cardCVVNo = value;}
 }
 public double Amount
 {
  get {return amount;}
  set { amount = value;}
 }
}
public interface Ypay
{
 string CustCardNo {get;set;}
 string CardOwnerName {get;set;}
 string CardExpMonthDate {get;set;}
```

```
  int CVVNo {get;set;}
  double TotalAmount {get;set;}
 }
 public class XpayToYpayAdapter : Ypay
 {
  private string custCardNo;
  private string cardOwnerName;
  private string cardExpMonthDate;
  private int cVVNo;
  private double totalAmount;
  private readonly Xpay xpay;
  public XpayToYpayAdapter(Xpay xpay)
  {
   this.xpay = xpay;
   setProp();
  }
  public  string CustCardNo
  {
   get{return custCardNo;}
   set{custCardNo = value;}
  }
  public string CardOwnerName
  {
   get{return cardOwnerName;}
   set{cardOwnerName = value;}
  }
  public string CardExpMonthDate
  {
   get{return cardExpMonthDate;}
   set{cardExpMonthDate = value;}
  }
  public int CVVNo
  {
   get{return cVVNo;}
   set{cVVNo = value;}
  }
  public double TotalAmount
  {
   get{return totalAmount;}
   set{totalAmount = value;}
  }
  private void setProp()
  {
   CardOwnerName = xpay.CustomerName;
   CustCardNo = xpay.CreditCardNo;
   CardExpMonthDate= xpay.CardExpMonth+"/"+ xpay.CardExpYear;
   CVVNo = xpay.CardCVVNo;
   TotalAmount = xpay.Amount;
  }
 }
}
```

3.3 Шаблон Композит (Composite)

Призначення

Композит (або Компонувальник) – це структурний патерн проектування, що дає можливість згрупувати декілька об’єктів у деревоподібну структуру, а потім працювати з нею так, ніби це одинарний об’єкт.

Компонувальник дозволяє зберігати деревоподібну структуру та працювати однаково з батьками та нащадками у дереві.

Композит може бути реалізований в будь-якому місці, де є ієрархічний характер системи або підсистеми й потрібно обробляти окремі об’єкти та композиції об’єктів однотипно. Структура композитного об’єкта полягає в тому, що один батько (композит) може мати багато нащадків, а кожен нащадок має лише одного батька.

Коли потрібно використовувати патерн

– Коли потрібно презентувати ієрархії об'єктів частково або в цілому.

– Коли потрібно, щоб клієнти могли ігнорувати різницю між композиціями об’єктів і окремими об’єктами. Клієнти будуть поводитися з усіма об’єктами композитної структури однаково.

Переваги

– Спрощує архітектуру клієнта при роботі з складним деревом компонентів.

– Полегшує додавання нових видів компонентів.

Недоліки

– Створює занадто загальний дизайн класів.

Загальна структура патерну

– Component визначає інтерфейс для всіх об’єктів в композиції як складених, так і кінцевих (листкових) вузлів. Компонент може реалізувати поведінку за замовчуванням для загальних методів.

– Composite – визначає поведінку компонентів, що мають нащадків, та зберігає дочірні компоненти. Composite також імплементує операції, пов’язані з листям (Leaf). Ці операції можуть мати або не мати ніякого сенсу – це залежить від функціональних реалізацій шаблону, що використовується.

– Leaf визначає поведінку елементів у композиції шляхом реалізації операцій, які підтримує Компонент. Leaf також успадковує методи, які не обов’язково мають певне значення для вузла лист.

– Client маніпулює об’єктами в композиції через інтерфейс компонентів.

На рисунку 3.4 подано UML-діаграму класів для даного патерну.

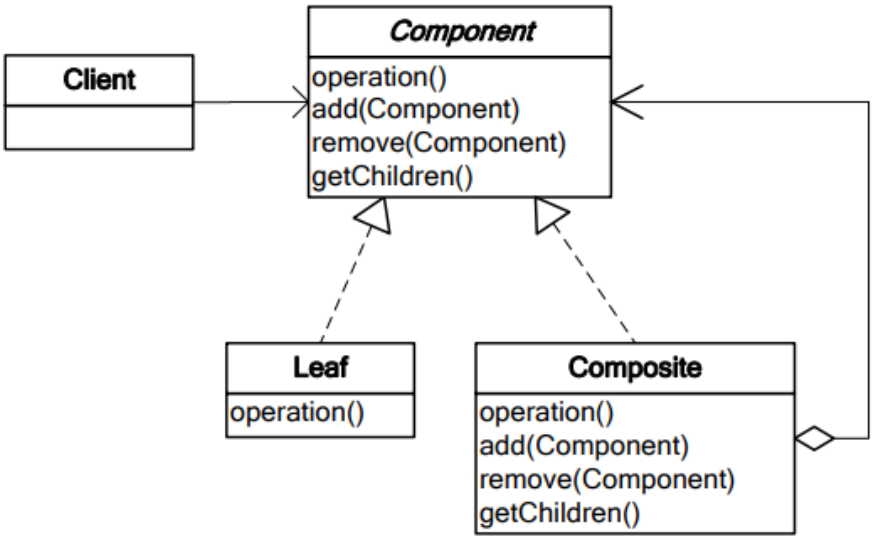


Рисунок 3.4 – UML-діаграма класів патерну Композит

Приклад

Умова. HTML-документ є ієрархічним за своєю суттю, починається з тега <html>, який є батьківським або кореневим тегом, і містить інші теги, які, в свою чергу, можуть бути батьківськими або дочірніми тегами.

Завдання. Реалізувати HTML-документ за допомогою шаблону Composite.

Перелік учасників даного прикладу

– Клас HtmlTag – це компонентний клас, який визначає всі методи, що застосовуються для композитного та класу Leaf. Є деякі методи, які мають бути спільними для обох розширених класів, і тому ці методи залишаються абстрактними в цьому класі, щоб забезпечити їх реалізацію у дочірніх класах.

Метод TagName() повертає ім’я тегу і його потрібно використовувати як з дочірніми (складеними) класами, так і з класом Leaf.

Методи StartTag та EndTag використовуються для встановлення початкового та кінцевого тегів html-елемента та мають бути реалізовані обома дочірніми класами, тому вони зберігаються абстрактними у цьому класі.

Метод generateHtml() – це операція, яка має підтримувати обидва розширені класи. Для простоти він просто виводить тег на консоль.

– Клас HtmlParentElement – це складений клас, який реалізує такі методи, як addChildTag, RemoveChildTag, getChildren, які мають бути реалізовані класом, щоб стати складовою структури.

– Клас HtmlElement – це клас листя, і його головне завдання – реалізувати метод операції, який у цьому прикладі є методом generateHtml. Він друкує startTag, tagBody, якщо він потрібен, та endTag дочірнього елемента.

Програмна реалізація

```
using System;
namespace Composite
{
 class Program
 {

  public static void Main(string[] args)
  {
   HtmlTag parentTag = new HtmlParentElement("<html>");
   parentTag.StartTag = "<html>";
   parentTag.EndTag = "</html>";
   HtmlTag p1 = new HtmlParentElement("<body>");
   p1.StartTag = "<body>";
   p1.EndTag = "</body>";
   parentTag.addChildTag(p1);
   HtmlTag child1 = new HtmlElement("<p>");
   child1.StartTag = "<p>";
   child1.EndTag = "</p>";
   child1.TagBody = "Testing html tag library";
   p1.addChildTag(child1);
   child1 = new HtmlElement("<p>");
   child1.StartTag = "<p>";
   child1.EndTag = "</p>";
   child1.TagBody = "Paragraph 2";
   p1.addChildTag(child1);
   parentTag.generateHtml();
   Console.ReadLine ( );
  }
 }
 public abstract class HtmlTag
 {
  public abstract string TagName {get;}
  public abstract string StartTag {set;}
  public abstract string EndTag {set;}
  public virtual string TagBody
  {
   set
   {
    throw new System.NotSupportedException("Current operation is not
support for this object");
   }
  }
  public virtual void addChildTag(HtmlTag htmlTag)
  {
   throw new System.NotSupportedException("Current operation is not
support for this object");
  }
  public virtual void removeChildTag(HtmlTag htmlTag)
  {
```

```
   throw new System.NotSupportedException("Current operation is not
support for this object");
  }
  public virtual IList<HtmlTag> Children
  {
   get
   {
    throw new System.NotSupportedException("Current operation is not
support for this object");
   }
  }
  public abstract void generateHtml();
 }
 public class HtmlParentElement : HtmlTag
 {
  private string tagName;
  private string startTag;
  private string endTag;
  private IList<HtmlTag> childrenTag;
  public HtmlParentElement(string tagName)
  {
   this.tagName = tagName;
   this.startTag = "";
   this.endTag = "";
   this.childrenTag = new List<HtmlTag>();
  }
  public override string TagName
  {
   get{return tagName;}
  }
  public override string StartTag
  {
   set{this.startTag = value;}
  }
  public override string EndTag
  {
   set{this.endTag = value;}
  }
  public override void addChildTag(HtmlTag htmlTag)
  {
   childrenTag.Add(htmlTag);
  }
  public override void removeChildTag(HtmlTag htmlTag)
  {
   childrenTag.Remove(htmlTag);
  }
  public override IList<HtmlTag> Children
  {
   get{return childrenTag;}
  }
  public override void generateHtml()
```

```
  {
   Console.WriteLine(startTag);
   foreach (HtmlTag tag in childrenTag){tag.generateHtml();}
   Console.WriteLine(endTag);
  }
 }
 public class HtmlElement : HtmlTag
 {
  private string tagName;
  private string startTag;
  private string endTag;
  private string tagBody;
  public HtmlElement(string tagName)
  {
   this.tagName = tagName;
   this.tagBody = "";
   this.startTag = "";
   this.endTag = "";
  }
  public override string TagName
  {
   get{return tagName;}
  }
  public override string StartTag
  {
   set{this.startTag = value;}
  }
  public override string EndTag
  {
   set{this.endTag = value;}
  }
  public override string TagBody
  {
   set{this.tagBody = value;}
  }
  public override void generateHtml()
  {
   Console.WriteLine(startTag + "" + tagBody + "" + endTag);
  }
 }
}
```

3.4 Шаблон Проксі (Proxy)

Призначення

Проксі (Proxy) – це структурний патерн проектування, що дає можливість підставляти замість реальних об’єктів спеціальні об’єкти-замінники. Ці об’єкти перехоплюють виклики до оригінального об’єкта, дозволяючи зробити щось до або після передачі виклику оригіналові.

Proxy підміняє реальний об’єкт та надсилає запити до нього тоді, коли це потрібно. Проксі також може ініціалізувати реальний об’єкт, якщо він до того не існував.

Існує кілька видів проксі, найпоширенішими з яких є нижчевказані.

Віртуальний проксі – передає створення об’єкта іншому об’єкту (корисно, якщо процес створення може бути повільним або виявиться непотрібним).

Захищений (аутентифікаційний) проксі – контролює доступ до оригінального об’єкта. Захищений проксі корисний, коли об’єкти мають різні права доступу.

Віддалений проксі – кодує запити та надсилає їх мережею.

Розумний проксі – додає або змінює запити, перш ніж надсилати їх.

Коли потрібно використовувати патерн

– Коли є важкий об’єкт, який завантажує дані з файлової системи або бази даних, можна заощадити ресурси й створити об’єкт тоді, коли він дійсно буде потрібним (віртуальний проксі).

– Коли в програмі є різні типи користувачів, і потрібно захистити об’єкт від неавторизованого доступу (захищальний або аутентифікаційний проксі).

– Коли справжній сервісний об’єкт знаходиться на віддаленому сервері (віддалений проксі).

– Коли потрібно зберігати історію звернень до сервісного об’єкта (логувальний проксі).

– Коли є потреба в кешуванні результатів запитів клієнтів і керуванні їхнім життєвим циклом (розумний проксі).

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

– Збільшує час отримання відклику від сервісу.

Загальна структура патерну

– Клас Subject – визначає загальний інтерфейс для RealSubject і Proxy, щоб проксі можна було використовувати там, де очікується RealSubject.

– Клас Proxy – клас, що підтримує посилання, яке дозволяє проксі отримувати доступ до реального об’єкта. Проксі може посилатися на Subject, якщо інтерфейси RealSubject і Subject однакові.

– Клас RealSubject – визначає реальний об’єкт, який є представником проксі.

– Request() – метод класу Subject, який спрямовується через проксі.

На рисунку 3.5 подано UML-діаграму класів для даного патерну.

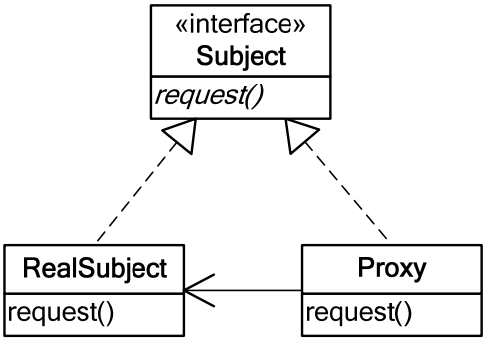


Рисунок 3.5 – UML-діаграма класів патерну Проксі

Приклад

Умова. Є неактивний реальний об’єкт, який може виконувати певну дію, але тільки в активному стані.

Завдання. Реалізувати активацію об’єкта двома способами: через віртуальний та захищений проксі.

Перелік учасників даного прикладу

– Інтерфейс ISubject – визначає загальний інтерфейс. Містить метод Request().

– Клас Subject – реалізує інтерфейс ISubject.

– Клас Proxy – реалізує інтерфейс Isubject, може створювати об’єкт Subject, якщо він раніше не був створений.

– Клас ProtectionProxy – реалізує інтерфейс Isubject, може створювати об’єкт Subject, якщо він раніше не був створений. При цьому виконує аутентифікацію.

Програмна реалізація

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Proxy
{
 class Program
 {
  static void Main(string[] args)
  {
   ISubject subject = new Proxy();
   Console.WriteLine(subject.Request());
   Console.WriteLine(subject.Request());
   ISubject subject1 = new ProtectionProxy();
```

```
   Console.WriteLine(subject1.Request());
   Console.WriteLine((subject1 as ProtectionProxy).
Authenticate("Secret"));
  Console.WriteLine((subject1 as ProtectionProxy).
Authenticate("krakazjablic"));
  Console.WriteLine(subject1.Request());
  Console.ReadLine();
 }
}
public interface ISubject
{
 string Request();
}
class Subject : ISubject
{
 public string Request()
 {
  return "\"реальний об'єкт виконує запит\" \n";
 }
}
public class Proxy : ISubject
{
 Subject subject;
 public string Request()
 {
  // A virtual proxy creates the object only on its first method call
  if (subject == null)
  {
   Console.WriteLine("Об'єкт неактивний");
   subject = new Subject();
  }
  Console.WriteLine("Об'єкт активний");
  return "Proxy: Виклик: " + subject.Request();
 }
}
public class ProtectionProxy : ISubject
{
 Subject subject;
 string password = "krakazjablic";

  public string Authenticate(string supplied)
  {
   if (supplied != password)
    return "Protection Proxy: В доступі відмовлено";
   else
    subject = new Subject();
   return "Protection Proxy: Аутентифіковано";
  }
```

```
  public string Request()
  {
   if (subject == null)
    return "Protection Proxy: Потрібна аутентифікація";
   else
    return "Protection Proxy: Виклик: " +
 subject.Request();
  }
 }
}
```

3.5 Шаблон Декоратор (Decorator)

Призначення

Декоратор – це структурний шаблон проектування, що дає можливість динамічно розширяти функціональність об’єктів без потреби зміни вихідного джерела класу або використання наслідування, загортаючи їх у корисні «обгортки».

Об’єкт Decorator призначений для того ж інтерфейсу, що і базовий об’єкт. Це дозволяє клієнтському об’єкту взаємодіяти з об’єктом Decorator так само, як і з базовим фактичним об’єктом.

Об’єкт Decorator містить посилання на фактичний об’єкт та отримує всі запити від клієнта. В свою чергу, він пересилає ці виклики до основного об’єкта.

Об’єкт Decorator додає деяку додаткову функціональність до або після переадресації запитів на базовий об’єкт. Це гарантує, що додаткова функціональність може бути додана даному об’єкту ззовні під час виконання без зміни його структури.

Коли потрібно використовувати патерн

– Коли потрібно додавати об’єктам нову функціональність динамічно, непомітно для коду, який її використовує.

– Коли немає можливості розширення функціональності об’єкта за допомогою наслідування або розширення через підкласи є недоцільним.

Переваги

– Запобігає розповсюдженню підкласів.

– Більша гнучкість, ніж у наслідування.

– Дозволяє додавати функціональність динамічно, при цьому дає можливість додавати кілька нових функціональностей одразу.

Недоліки

– Велика кількість малих класів.

– Складність конфігурування об’єктів, які загорнуто в декілька обгорток одночасно.

Загальна структура патерну

– Клас Component – інтерфейс або абстрактний клас, який визначає інтерфейс для успадкованих об’єктів.

– Клас ConcreteComponent – конкретна реалізація компонента, в яку за допомогою декоратора додається нова функціональність.

– Клас Decorator – реалізується у вигляді абстрактного класу і має той же базовий клас, що й об’єкти, які декоруються. Тому базовий клас Component має бути, по можливості, легким і визначати тільки базовий інтерфейс. Клас Decorator також зберігає посилання на об’єкт, що декорується, у вигляді об’єкта базового класу Component і реалізує зв’язок з базовим класом як через наслідування, так і через відношення агрегації.

– Клас ConcreteDecorator – містить додаткову функціональність, яка буде розширювати об’єкт.

На рисунку 3.6 подано UML-діаграму класів для даного патерну.

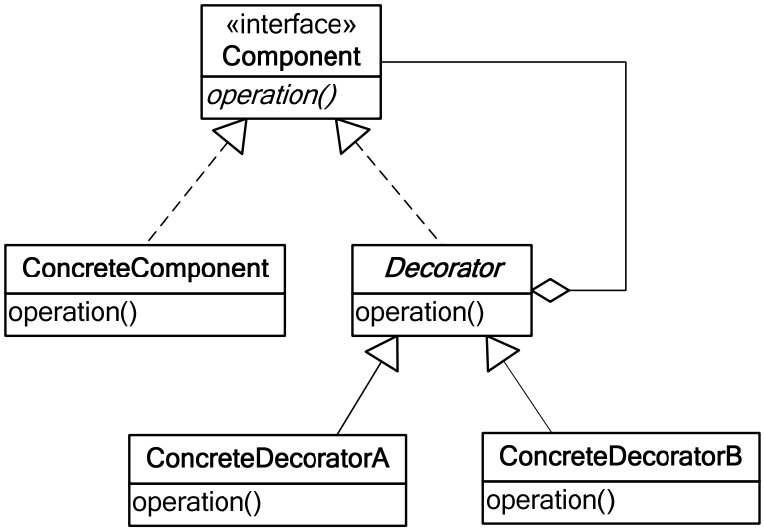


Рисунок 3.6 – UML-діаграма класів патерну Декоратор

Приклад

Умова. Працює піцерія, в якій можна виготовити власну піцу, використовуючи базову та вибрати різні інгредієнти до неї, наприклад м’ясо, помідори та кукурудзу.

Завдання. Реалізувати конструктор для піци, який видає готову піцу залежно від вимог клієнта, реалізувавши інгредієнти як декоратори.

Перелік учасників даного прикладу

– Інтерфейс Pizza – визначає властивості, що застосовуються до базового об’єкту. Визначає опис та вартість піци.

– Клас SimplyVegPizza – реалізує інтерфейс Pizza.

– PizzaDecorator – абстрактний клас декораторів, який має базовий клас Pizza та реалізує зв’язок з ним. Він буде розширюватись конкретними декораторами.

– Класи GreenOlives, RomaTomatoes, Spinach – класи конкретних декораторів (інгредієнтів).

Програмна реалізація

```
using System;
namespace Decorator
{
 class Program
 {
  public static void Main(string[] args)
  {
   Pizza pizza = new BasePizza();
   pizza = new Tomatoes(pizza);
   pizza = new Meat(pizza);
   pizza = new SweetCorn(pizza);
   Console.WriteLine("Description: " + pizza.Desc);
   Console.WriteLine("Price: " + pizza.Price.ToString());
   Console.ReadLine();
  }
 }
 public interface Pizza
 {
  string Desc { get; }
  double Price { get; }
 }
 public class BasePizza : Pizza
 {
  public string Desc
  {
   get {return "BasePizza (120)";}
  }
  public double Price
  {
   get {return 120;}
  }
 }
 public abstract class PizzaDecorator : Pizza
 {
  public abstract string Desc { get; }
  public abstract double Price { get; }
 }
 public class Meat : PizzaDecorator
 {
  private readonly Pizza pizza;
  public Meat(Pizza pizza)
  {
   this.pizza = pizza;
  }
  public override string Desc
  {
   get {return pizza.Desc + ", М'ясо (25)";}
```

```
  }
  public override double Price
  {
   get { return pizza.Price + 25;}
  }
 }
 public class Tomatoes : PizzaDecorator
 {
  private readonly Pizza pizza;
  public Tomatoes(Pizza pizza)
  {
   this.pizza = pizza;
  }

  public override string Desc
  {
   get { return pizza.Desc + ", Помідори (15.0)";}
  }
  public override double Price
  {
   get { return pizza.Price + 15;}
  }
 }
 public class SweetCorn : PizzaDecorator
 {
  private readonly Pizza pizza;
  public SweetCorn(Pizza pizza)
  {
   this.pizza = pizza;
  }
  public override string Desc
  {
   get { return pizza.Desc + ", Кукурудза (17)";}
  }
  public override double Price
  {
   get { return pizza.Price + 17;}
  }
 }
}
```

3.6 Шаблон Фасад (Facade)

Призначення

Фасад – це структурний шаблон проектування, який забезпечує простий інтерфейс до складної системи класів, бібліотеки або фреймворку.

Фасад дає єдину «точку доступу» до підсистеми, тим самим спрощуючи її використання та розуміння.

Фасад не інкапсулює класи або інтерфейси підсистеми, а просто забезпечує спрощений інтерфейс до їх функціональності. Клієнт може отримати доступ до цих класів безпосередньо.

Коли потрібно використовувати патерн

– Коли потрібно надати простий або спрощений інтерфейс до складної підсистеми.

– Коли потрібно розкласти підсистему на окремі рівні.

– Коли потрібно зменшити кількість залежностей між клієнтом і складною системою.

Переваги

– Ізолює клієнтів від компонентів складної підсистеми.

Недоліки

– Фасад прив’язаний до всіх класів програми.

Загальна структура патерну

– Клас Facade – перенаправляє запити клієнтів, іншим об’єктам, з яких складається підсистема.

– Subsystem classes – класи підсистеми, які реалізують фактичну функціональність системи. Вони нічого не знають про фасад (не мають посилань на фасад).

– Клієнт має доступ лише до класу Facade, щоб отримати функціональні можливості класів підсистеми.

На рисунку 3.7 подано UML-діаграму класів для даного патерну.

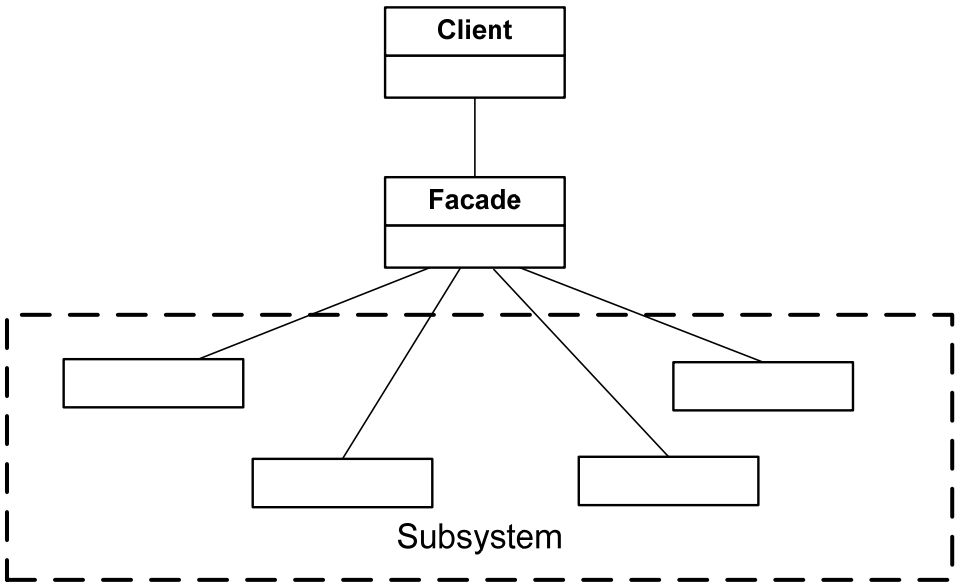


Рисунок 3.7 – UML-діаграма класів патерну Фасад

Приклад

Умова. Існує фреймворк для роботи з замовленнями, недоліком якого є необхідність роботи з багатьма різними класами.

Завдання. Реалізувати єдиний метод для оплати та доставки замовлень, який буде налаштовувати потрібні об’єкти фреймворку для отримання потрібного результату.

Перелік учасників даного прикладу

– Клас Packaging – клас, в якому реалізовано методи для різних видів пакування замовлення.

– Клас Payment – клас, в якому реалізовано методи для різних видів оплати замовлення.

– Клас Logistic – клас, в якому реалізовано метод для адреси, за якою потрібно доставити замовлення.

Ці три класи є класами підсистеми.

– Facade – клас, який надає інтерфейс для роботи клієнта з класами підсистеми.

– Client – клас клієнта, який взаємодіє з компонентами підсистеми та формує замовлення через клас Facade.

Програмна реалізація

```
using System;
namespace facade
{
 public class Client
 {
  public static void Main()
  {
   Console.OutputEncoding = System.Text.Encoding.UTF8;
   Facade facade = new Facade(new Payment(), new Packaging(), new
Logistic());
   Console.WriteLine("Замовлення 1. Оплата готівкою:");
   facade.OrderCash();
   Console.WriteLine("Замовлення 2. Оплата банківською картою:");
   facade.OrderCard();
  }
 }
 public class Packaging
 {
  public void Box()
  {
   Console.WriteLine("Запаковано в коробку");
  }
  public void Package()
```

```
  {
   Console.WriteLine("Запаковано в пакет");
  }
 }
 public class Payment
 {
  public void Cash()
  {
   Console.WriteLine("Оплочено готівкою");
  }
  public void Card()
  {
   Console.WriteLine("Оплочено карткою");
  }
 }
 public class Logistic
 {
  public void Address()
  {
   Console.WriteLine("Хмельницьке шосе, 95");
  }
 }
 public class Facade
 {
  Payment payment;
  Packaging pack;
  Logistic log;
  public Facade(Payment a, Packaging b, Logistic c)
  {
   payment = a;
   pack = b;
   log = c;
 }
 public void OrderCash()
 {
   payment.Cash();
   pack.Box();
   log.Address();
  }
  public void OrderCard()
  {
   payment.Card();
   pack.Package();
   log.Address();
  }
 }
}
```

3.7 Шаблон Легковаговик (Flyweight)

Призначення

Легковаговик – це структурний патерн проектування, що дозволяє 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.

Легковаговик – це спільний об’єкт, який можна використовувати в кількох контекстах одночасно. Ключовим поняттям тут є відмінність між внутрішнім і зовнішнім станами. Внутрішній стан – це незмінні дані об’єкта, які не залежать від контексту, а всі інші дані є зовнішнім станом.

Коли потрібно використовувати патерн

– Коли програма використовує велику кількість об’єктів.

– Коли витрати пам’яті для зберігання високі через велику кількість об’єктів.

– Коли більшість станів об’єктів можна зробити зовнішніми.

– Коли багато груп об’єктів можуть бути замінені відносно невеликою кількістю спільних об’єктів після видалення зовнішнього стану.

– Коли програма не залежить від ідентичності об’єкта.

Переваги

– Дозволяє заощадити оперативну пам’ять пристрою.

Недоліки

– Рідко використовується на практиці.

– Неефективне використання часу роботи процесора для пошуку або обчислення контексту.

Загальна структура патерну

– Flyweight – оголошує інтерфейс, за допомогою якого легковаговики можуть приймати та виконувати дії у зовнішньому стані.

– ConcreteFlyweight – реалізує інтерфейс Flyweight та додає сховище для внутрішнього стану, якщо потрібно. Об’єкт ConcreteFlyweight має бути загальнодоступним. Будь-який стан, який він зберігає, має бути внутрішнім, тобто незалежним від контексту об’єкта ConcreteFlyweight.

– FlyweightFactory – створює і керує об’єктами легковаговиками. Забезпечує належний загальний доступ легковаговиків. Коли клієнт запитує легковаговик, об’єкт FlyweightFactory надає наявний екземпляр або створює його, якщо такого немає.

– Client – зберігає посилання на легковаговиків, обчислює або зберігає їх зовнішній стан.

UML-діаграму класів патерну Легковаговик наведено на рис. 3.8.

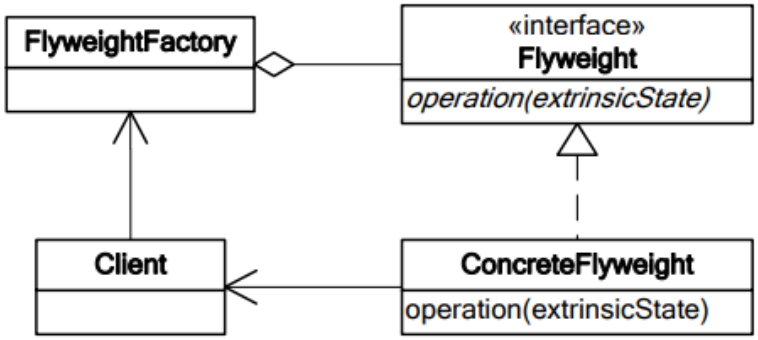


Рисунок 3.8 – UML-діаграма класів патерну Легковаговик

Програмна реалізація

```
using System;
using System.Collections;
namespace Flyweight
{
 class Program
 {
  static void Main()
  {
   string msg = "AAAZBBZZB";
   char[] mch = msg.ToCharArray();
   CharacterFactory factory = new CharacterFactory();
   int pntSize = 10;
   foreach (char ch in mch)
   {
    pntSize++;
    Character charact = factory.GetCharacter(ch);
    charact.DisplayCharacter(pntSize);
   }
   Console.Read();
  }
 }
 class CharacterFactory
 {
  private Hashtable charsTable = new Hashtable();
  public Character GetCharacter(char ch)
  {
   Character charact = charsTable[ch] as Character;
   if (charact == null)
   {
    switch (ch)
    {
     case 'A':
      charact = new CharacterA();
      break;
     case 'B':
      charact = new CharacterB();
      break;
```

```
      case 'Z':
       charact = new CharacterZ();
       break;
     }
     charsTable.Add(ch, charact);
    }
    return charact;
   }
  }
  abstract class Character
  {
   protected char charSymbol;
   protected int charWidth;
   protected int charHeight;
   protected int charAscent;
   protected int charDescent;
   protected int pntSize;
   public abstract void DisplayCharacter(int pointSize);
  }
  class CharacterA : Character
  {
   public CharacterA()
   {
    this.charSymbol = 'A';
    this.charHeight = 100;
    this.charWidth = 120;
    this.charAscent = 70;
    this.charDescent = 0;
   }
   public override void DisplayCharacter(int pointSize)
   {
    this.pntSize = pointSize;
    Console.WriteLine("Symbol " + this.charSymbol + " (point size = "
+ this.pntSize + ")");
   }
  }
  class CharacterB : Character
  {
  public CharacterB()
  {
   this.charSymbol = 'B';
   this.charHeight = 100;
   this.charWidth = 140;
   this.charAscent = 72;
   this.charDescent = 0;
  }
  public override void DisplayCharacter(int pointSize)
  {
   this.pntSize = pointSize;
   Console.WriteLine("Symbol "+charSymbol+"(point size="+pntSize+
")");
```

```
  }
 }
 class CharacterZ : Character
 {
  public CharacterZ()
  {
   this.charSymbol = 'Z';
   this.charHeight = 100;
   this.charWidth = 100;
   this.charAscent = 68;
   this.charDescent = 0;
  }
  public override void DisplayCharacter(int pointSize)
  {
   this.pntSize = pointSize;
   Console.WriteLine("Symbol " + this.charSymbol + " (point size = "
+ this.pntSize + ")");
  }
 }
}
```

Питання для самоперевірки

1. Коротко охарактеризуйте структурні шаблони проектування.
2. Наведіть переваги та недоліки використання шаблону Міст.
3. Наведіть UML-діаграму для шаблону Фасад.
4. Коли доцільно використовувати шаблон Адаптер?
5. Охарактеризуйте шаблон Міст. Наведіть приклад, пов’язаний з реальним життям.
6. Охарактеризуйте які можливості дає використання шаблону Декоратор. Наведіть приклад з реального життя.
7. Поясніть в який випадках використовується патерн Композит. Наведіть його переваги та недоліки.
8. Поясніть переваги використання шаблону Міст порівняно з породжувальним шаблоном Абстрактна Фабрика.
9. Наведіть UML-діаграму для шаблону Адаптер.
10. Охарактеризуйте шаблон Проксі.
11. Поясніть в який випадках використовується патерн Фасад. Наведіть його переваги та недоліки.
12. Наведіть UML-діаграму для шаблону Міст.
13. Наведіть UML-діаграму для шаблону Будівельник.
14. Поясніть призначення класів, що використовуються для шаблону Проксі.
15. Наведіть UML-діаграму для шаблону Декоратор.

4 ШАБЛОНИ ПОВЕДІНКИ

Патерни поведінки (behavioral patterns) використовуються для передачі управління в системі. Їх застосування дозволяє досягти значного підвищення як ефективності системи, так і зручності її експлуатації. Патерни поведінки пов'язані з алгоритмами та розподілом відповідальності між об'єктами, вони описують не лише шаблони об'єктів або класів, а й шаблони зв'язків між ними.

Спостерігач визначає залежності (один до багатьох) між об'єктами таким чином, що коли один об'єкт змінює стан, всі його відношення змінюються автоматично. **Ітератор** надає спосіб послідовного доступу до елементів сукупних об'єктів, не розкриваючи їх базове подання. **Стратегія** визначає сімейство алгоритмів, інкапсулює кожен з них і робіть їх взаємозамінними, а також дозволяє алгоритму змінюватися незалежно від клієнтів, які його використовують. **Знімок**, не порушуючи інкапсуляції, захоплює внутрішній стан об'єкта, щоб потім об'єкт можна було відновити до цього стану. **Стан** дозволяє об'єкту змінити поведінку, коли змінюється його внутрішній стан, при цьому об'єкт буде змінювати свій клас. **Шаблонний метод** визначає структуру алгоритму, передає деякі кроки до підкласів, а також дозволяє підкласам перевизначати певні кроки алгоритму без зміни структури алгоритму. **Ланцюжок обов'язків** уникає зв'язку між відправником запиту та його одержувачем, даючи можливість обробити запит більш ніж одному об'єкту; він об'єднує об'єкти-одержувачі в ланцюжок і передає запит по ланцюжку, поки об'єкт не обробить його. **Посередник** визначає об'єкт, який інкапсулюється як взаємодія набору об'єктів; не дає об'єктам чітко посилатися один на одного, і дозволяє змінювати їх взаємодію незалежно один від одного. **Відвідувач** дозволяє визначити нову операцію без зміни класів елементів, з якими вона буде працювати. **Команда** інкапсулює запит на виконная певної дії як об'єкт. **Інтерпретатор** визначає граматику певної мови для конкретної проблемної області.

4.1 Шаблон Observer (Спостерігач)

Призначення

Призначений для реалізації архітектури «Підписання-Розповсюдження» («Subscribe-Publish»), яка полягає у тому, що розповсюджувач (об'єкт класу Publisher) розсилає повідомлення всім підписникам (об'єктам класу Subscriber), які були підписані на це розсилання. У даному патерні розповсюджувач називається також суб'єктом (об'єктом класу Subject), а підписник називається також спостерігачем (об'єктом класу Observer).

Для реалізації даної архітектури клас Publisher використовує делегат, до якого «причіплюються» функції об'єктів класу Subscriber. Суть схеми полягає у тому, що, з одного боку, існує об'єкт, який реалізує «розповсюдження», ініціюючи виконання свого делегата, а з іншого – є об'єкти, що «підписуються» на це «розповсюдження», додаючи свої функції до даного делегата. Для цього розповсюджувач має надавати свій делегат у відкритий доступ, який можуть використовувати функції «підписування» і «відписування» підписника.

Однак у даному випадку делегати напряму не використовуються, оскільки у такому призначенні вони не є безпечними. Небезпека полягає у тому, що черговий клас-«підписник» може відмінити попередні «підписки», виконавши не додавання, а присвоєння своєї функції делегату. Тому «розповсюджувач» має надавати у відкритий доступ не сам делегат, а певні функції, що дозволяють виконувати лише додавання і віднімання функцій до делегата. Використання функцій «підписування» і «відписування» класами-«підписниками», з одного боку, і функцій додавання і віднімання до делегата класом-«розповсюджувачем», з іншого, робить всю схему «підписування-розповсюдження» дуже громіздкою. Для спрощення даної схеми створено спеціальний тип змінної, що називаються подією (event). Фактично подія є «обгорткою» навколо делегата, яка робить його безпечним, оскільки «підписними» можуть виконувати над подією лише операції + = і – =, тобто лише додати або відняти функцію. Це робить простою схему «підписування-розповсюдження».

Подія – це тип, що містить захищений делегат заданого типу, над яким можна виконувати лише операції + = і – =. Змінна події оголошується за допомогою ключового слова event із вказанням типу делегата, для якого вона створюється, наприклад:

```
…
delegate void MyFunk();
class A
{
 public event MyFunk ev;
}
void F1(){…}
void F2(){…}
static void Main()
{
 A a=new A();
 a.ev+=F2;
 a.ev+=F1;
 a.ev-=F1;
 a.ev();
}
…
```

Список учасників даного патерну

На рисунку 4.1 подано UML-діаграму класів патерну «Спостерігач» з використанням делегата і події.

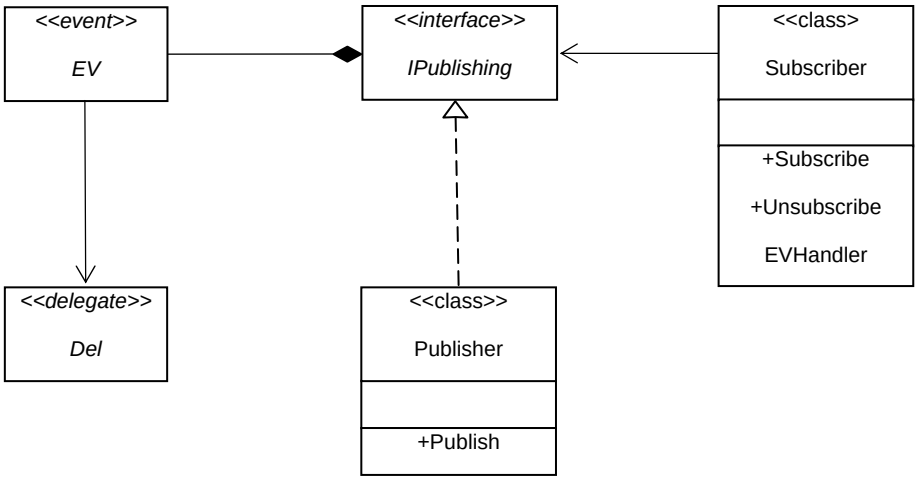


Рисунок 4.1 – UML-діаграма класів патерну «Спостерігач»

На діаграмі показано інтерфейс IPublishing, що містить подію EV, яка є обгорткою делегата Del. Клас Publisher реалізує цей інтерфейс і містить функцію Publish, яка ініціює подію EV, якщо відбулось хоча б одне підписання. Для підписування і відписування даний клас надає подію у відкритому доступі. Клас Subscriber має функцію підписування Subscribe, функцію відписування Unsubscribe і обробник події.

Програмна реалізація

Приклад програми з використанням події за схемою «підписування-розповсюдження»:

```
using System;
namespace ConsoleApp1
{
 delegate void Del(string message);
 interface IPublishing
 {
  event Del EV;
 }
 class Publisher:IPublishing
 {
  public event Del EV = null;
  public void Publish(string msg)
  {
   if (EV != null)
    EV(msg);
  }
```

```
 }
 class Subscriber
 {
  string objName;
  public Subscriber(string name) { objName = name; }
  public void Subscribe(IPublishing ip)
  {
   ip.EV += EVHandler;
  }
  public void Unsubscribe(IPublishing ip)
  {
   ip.EV -= EVHandler;
  }
  public void EVHandler(string msg)
  {
   Console.WriteLine(objName+". Publisher say: " + msg);
  }
 }
 class Program
 {
  static void Main()
  {
   Publisher publisher = new Publisher();
   Subscriber subscriber1 = new Subscriber("Subscriber1");
   Subscriber subscriber2 = new Subscriber("Subscriber2");
   subscriber1.Subscribe(publisher);
   subscriber2.Subscribe(publisher);
   publisher.Publish("Good morning!");
   Console.WriteLine();
   subscriber2.Unsubscribe(publisher);
   publisher.Publish("Good day!");
  }
 }
}
```

З діаграми на рис. 4.1 видно, що класи Publisher і Subscriber не залежать один від одного. Проте повністю уникнути жорстких зв'язків у даній схемі неможливо. Обидва класи залежать від типу делегата. Для вирішення даної проблеми Microsoft пропонує в усіх випадках використовувати один і той самий тип делегата, який нічого не повертає і має два параметри: object sender – об'єкт узагальненого типу, що посилає повідомлення, і EventArgs e – посилання узагальненого типу на дані, які передає подія.

Разом з використанням сучасного інструментарію мови C# це дозволяє спростити код і зменшити залежність від типів. Наприклад:

```
using System;
namespace ConsoleApp32
{
 internal class Program
```

```
{
 static void Main(string[] args)
 {
  Publisher publisher = new Publisher();
  Subscriber subscriber = new Subscriber();
  subscriber.Subscribe(publisher);
  publisher.Publish("Hello from Publisher!");
 }
}
interface IEvent
{
 event EventHandler<string> ev;
}
class Publisher:IEvent
{
 public event EventHandler<string> ev;
 public void Publish(string msg)
 {
  ev?.Invoke(this,msg);
 }
}
class Subscriber
{
 public void Subscribe(IEvent iv)
 {
  iv.ev += (sender,msg) => Console.WriteLine($"Subscriber:
{sender.GetType().Name} say \"{msg}\"");
 }
}
}
```

4.2 Шаблон Ітератор (Iterator)

Призначення

Ітератор – це поведінковий патерн проектування, який надає спосіб послідовного доступу до елементів складеного об’єкта (контейнера), не розкриваючи його базового подання. Термін «контейнер» можна визначити як набір даних або об’єктів. Об’єкти всередині контейнера, в свою чергу, можуть бути колекціями, що робить його колекцією колекцій. Потрібно відмітити, що одну і ту ж колекцію можна обходити різними способами.

Об’єкт Container має бути розроблений для забезпечення загальнодоступного інтерфейсу у вигляді об’єкта Iterator для різних об’єктів клієнта для доступу до його вмісту. Об’єкт Iterator містить відкриті методи, які дозволяють клієнтському об’єкту переміщатися за списком об’єктів у контейнері. Один і той самий контейнер можуть обходити декілька ітераторів.

Коли потрібно використовувати патерн

– Коли потрібно отримати доступ до вмісту складеного об’єкта, не розкриваючи його внутрішнього подання.

– Коли потрібно забезпечити множинні обходи складеного об’єкта.

– Коли потрібно забезпечити єдиний інтерфейс для обходу різних колекцій.

Переваги

– Дозволяє одночасно обходити колекцію у різних напрямках.

– Дозволяє реалізувати різні способи обходу колекції.

Недоліки

– Не завжди виправдане його використання.

Загальна структура патерну

– Iterator – визначає інтерфейс для доступу та проходу елементів.

– ConcreteIterator – реалізує інтерфейс Iterator та відстежує поточну позицію в проходженні складеного об’єкта.

– Aggregate (Сукупність) – визначає інтерфейс для створення об’єкта Iterator.

– ConcreteAggregate – реалізує інтерфейс створення об’єкта Iterator, щоб повернути екземпляр відповідного ConcreteIterator.

UML-діаграму класів патерну Ітератор подано на рис. 4.2.

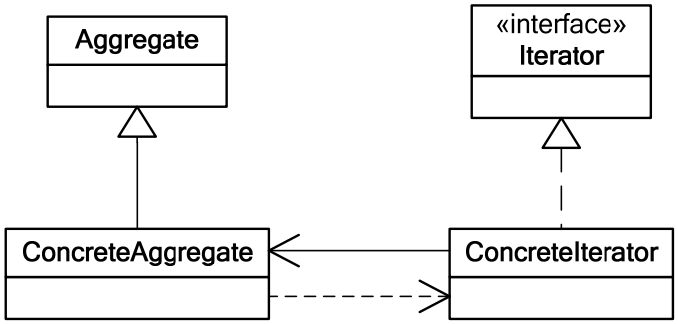


Рисунок 4.2 – UML-діаграма класів патерну Ітератор

Ітератор може бути розроблений або як внутрішній ітератор, або як зовнішній ітератор.

Внутрішні ітератори

– Сама колекція пропонує методи, що дозволяють клієнту відвідувати різні об’єкти в колекції.

– В колекції може бути лише один ітератор у будь-який момент часу.

– Колекція має підтримувати або зберігати стан ітерації.

Зовнішні ітератори

– Функція ітерації відокремлена від колекції та зберігається всередині іншого об’єкта, який називається ітератором. Зазвичай сама колекція пове-

ртає клієнту відповідний об’єкт ітератора залежно від введення клієнта.

– У певній колекції може бути кілька ітераторів у будь-який момент часу.

Приклад

Умова. Задано список студентів.

Завдання. Реалізувати виведення в консоль імен студентів згідно зі списком за допомогою шаблону Ітератор.

Перелік учасників даного прикладу

– Інтерфейс IIterator – визначає інтерфейс для доступу до об’єктів (студентів).

– Інтерфейс IContainer – визначає інтерфейс для отримання ітератора.

– Клас Student – допоміжний клас, в якому містяться дані про ім’я студента.

– Клас Group – реалізує інтерфейс IContainer та створює масив об’єктів – список студентів.

– Клас GroupIterator – реалізує інтерфейс IIterator.

Програмна реалізація

```
using System;
using System.Collections;
namespace Iterator
{
 class Program
 {
  public static void Main(string[] args)
  {
   IContainer group = new Group();
   IIterator iterator = group.GetIterator();
   while (iterator.HasNext())
    Console.WriteLine(((Student) iterator.Next()).Name);
   Console.ReadLine();
  }
 }
 interface IIterator
 {
  bool HasNext();
  object Next();
 }
 interface IContainer
 {
  IIterator GetIterator();
 }
 class Student
 {
  public string Name { get; }
```

```
  public Student(string name) {Name = name;}
 }
 class Group : IContainer
 {
   public readonly Student[] Students =
   {
    new Student("Mark"),
    new Student("John"),
    new Student("Dave")
   };
   public IIterator GetIterator()
   {
    return new GroupIterator(this);
   }
  }
  class GroupIterator : IIterator
  {
   private int _index;
   private readonly Group _group;
   public GroupIterator(Group group)
   {
    _group = group;
   }
   public bool HasNext()
   {
    return _index < _group.Students.Length;
   }
   public object Next() {
   return HasNext() ? _group.Students[_index++] : null;
  }
 }
}
```

4.3 Шаблон Стратегія (Strategy)

Призначення

Стратегія – це поведінковий патерн проектування, який визначає сімейство алгоритмів, інкапсулює кожен з них та дає можливість заміняти алгоритми один на інший прямо під час виконання програми. Стратегія дозволяє алгоритму змінюватися незалежно від клієнтів, які його використовують. Патерн забезпечує гнучкість програмного коду.

Коли потрібно використовувати патерн

– Коли потрібно використовувати різні варіанти алгоритму всередині одного об’єкта.

– Коли потрібно налаштувати клас з одним із багатьох видів поведінки, якщо є багато споріднених класів, які відрізняються лише своєю поведінкою.

– Коли потрібно уникнути розкриття складних структур даних, специфічних для алгоритмів, якщо алгоритм використовує дані, про які клієнти не мають знати.

– Коли клас визначає багато способів поведінки, які відображаються як кілька умовних операторів у його операціях, доцільно перемістити пов'язані умовні гілки до власного класу Strategy.

Переваги

– Дозволяє змінювати алгоритм в процесі виконання програми.

– Дозволяє ізолювати дані алгоритму від інших класів.

– Дозволяє замінити наслідування делегуванням

Недоліки

– Клієнту потрібно знати різницю між стратегіями, щоб вибрати потрібну.

Загальна структура патерну

– Інтерфейс Strategy – оголошує інтерфейс, спільний для всіх підтримуваних алгоритмів.

– Клас ConcreteStrategy – реалізує алгоритм за допомогою інтерфейсу Strategy.

– Клас Context використовує цей інтерфейс для виклику алгоритму, визначеного класом ConcreteStrategy. Context конфігурується об'єктом ConcreteStrategy. Підтримує посилання на об'єкт Strategy. Може визначати інтерфейс, який дозволяє Strategy отримувати доступ до своїх даних.

UML-діаграму класів патерну Стратегія подано на рис. 4.3.

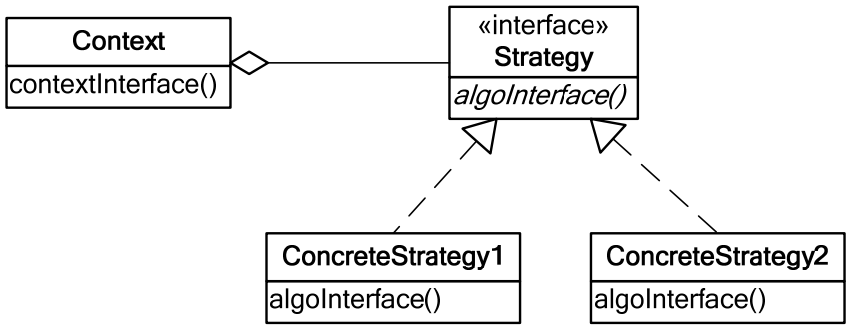


Рисунок 4.3 – UML-діаграма класів патерну Стратегія

Приклад

Умова. Задано текстовий редактор, який має різні засоби форматування тексту. Робота такого редактора полягає в тому, щоб передати текст у форматувальник для його форматування.

Завдання. Реалізувати форматувальник тексту для текстового редактора. Створити різні засоби форматування тексту (наприклад, перетворення тексту в формат верхнього та нижнього регістрів) як окремі стратегії. За-

безпечити можливість передачі потрібного засобу текстовому редактору, щоб останній міг відформатувати текст відповідно до вимог.

Перелік учасників даного прикладу

– TextFormatter – інтерфейс, який реалізується всіма конкретними форматувальниками.

– Клас CapTextFormatter – конкретний форматувальник тексту, який реалізує інтерфейс TextFormatter, використовується для перетворення тексту в форматі верхнього регістру.

– Клас LowerTextFormatter – конкретний форматувальник тексту, який реалізує інтерфейс TextFormatter, використовується для перетворення тексту в форматі нижнього регістру.

– клас TextEditor містить посилання на інтерфейс TextFormatter. Клас містить метод publishText, який пересилає текст у форматувальник, щоб опублікувати текст у потрібному форматі.

Програмна реалізація

```
using System;
namespace Strategypattern
{
 public class TestStrategyPattern
 {
  public static void Main(string[] args)
  {
   Console.OutputEncoding = System.Text.Encoding.UTF8;
   TextFormatter formatter = new CapTextFormatter();
   TextEditor editor = new TextEditor(formatter);
   editor.publishText("Тестування тексту в форматі великих літер");
   formatter = new LowerTextFormatter();
   editor = new TextEditor(formatter);
   editor.publishText("Тестування ТЕКСТУ В ФОРМАТІ МАЛИХ ЛІТЕР");
   Console.ReadLine ( );
  }
 }
 public interface TextFormatter
 {
  void format(string text);
 }
 public class CapTextFormatter : TextFormatter
 {
  public void format(string text)
  {
   Console.WriteLine("[CapTextFormatter]: " + text.ToUpper());
  }
 }
 public class LowerTextFormatter : TextFormatter
 {
  public void format(string text)
```

```
  {
   Console.WriteLine("[LowerTextFormatter]: " + text.ToLower());
  }
 }
 public class TextEditor
 {
  private readonly TextFormatter textFormatter;
  public TextEditor(TextFormatter textFormatter)
  {
   this.textFormatter = textFormatter;
  }
  public virtual void publishText(string text)
  {
   textFormatter.format(text);
  }
 }
}
```

4.4 Шаблон Знімок (Memento)

Призначення

Знімок – це поведінковий патерн проектування, який дозволяє зберігати та відновлювати попередній стан об'єктів, не розкриваючи їх внутрішньої структури. Об'єкти, зазвичай, інкапсулюють певний або весь їх стан, роблячи його недоступним для інших об'єктів і неможливим для зберігання зовні. Розкриття цього стану може порушити інкапсуляцію, що може поставити під загрозу надійність та розширюваність програми. Патерн Memento може вирішити це, не розкриваючи внутрішньої структури об'єкта.

Коли потрібно використовувати патерн

– Коли потрібно зберегти стан (або частину стану) об'єкта, щоб відновити його пізніше.

– Коли прямий інтерфейс для отримання стану відкриє деталі реалізації та порушить інкапсуляцію об'єкта.

Переваги

– Не порушує інкапсуляцію основного об'єкта.

– Спрощує структуру основного об'єкта та йому не треба зберігати історію версій свого стану.

Недоліки

– Потребує багато пам'яті, якщо клієнти часто створюють знімки.

– Деякі мови програмування не гарантують наявності доступу до стану знімка тільки у основного об'єкта.

Загальна структура патерну

– Клас Originator – основний об'єкт. Створює знімок, що містить його поточний внутрішній стан та використовує його для відновлення внутріш-

нього стану. Вирішує, скільки знімків потрібно зберігати в будь-який момент часу.

– Клас Memento – зберігає внутрішній стан об'єкта Originator. Він може зберігати стільки внутрішніх станів об'єкта Originator, скільки потрібно його розробнику. Не дозволяє іншим класам, окрім Originator, отримати доступ до цього стану.

Memento має два інтерфейси.

Широкий інтерфейс до Originator, що дозволяє йому отримувати доступ до всіх даних, які потрібно зберегти чи відновити їх попередній стан.

Вузький інтерфейс до Caretaker, який може зберігати та передавати посилання на знімок, але не більше.

– Клас CareTaker – відповідає за збереження знімка. Ніколи не працює з знімком та не вивчає його вміст.

UML-діаграму класів патерну Знімок подано на рис. 4.4.

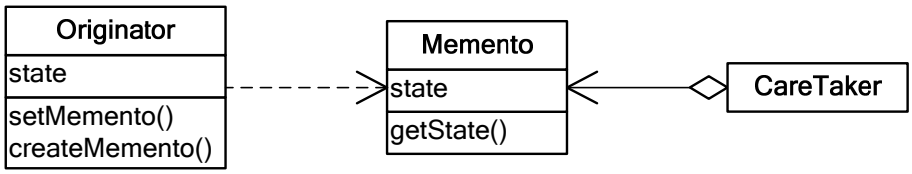


Рисунок 4.4 – UML-діаграма класів патерну Знімок

Приклад

Умова. Задано клас, який містить два поля дійсного типу X та Y.

Завдання. Потрібно виконати з цим класом різні математичні операції, дати користувачам операцію скасування. Користувач має мати можливість викликати операцію скасування, яка відновить стан об'єкта до певної збереженої точки. Також користувач має мати можливість у будь-який момент повернутися до початкових значень полів. Крім того, потрібно передбачити засіб, який користувач зможе використати для збереження стану об'єкта.

Перелік учасників даного прикладу

– Клас Originator – клас, стан об'єкта якого потрібно зберегти в пам'яті. Клас містить два поля дійсного типу X і Y, та має посилання на CareTaker, який використовується для збереження та відновлення збережених об'єктів, які визначають стан об'єкта Originator. Метод createSavepoint в конструкторі зберігає початковий стан об'єкта. Змінна lastUndoSavepoint використовується для збереження імені ключа останнього збереженого об'єкта для реалізації операції скасування.

Клас містить три типи скасування операцій: метод undo без параметрів – відновлює останній збережений стан; метод undo з іменем точки збере-

ження як параметром – відновлює стан, збережений за допомогою конкретного імені точки збереження; метод undoAll – просить опікуна очистити всі точки збереження та встановити його у початковий стан на момент створення об’єкта.

– Клас Memento використовується для зберігання стану Originator і зберігається опікуном.

– Клас CareTaker – це клас опікуна, який використовується для збереження та відновлення запитуваного об’єкта memento. Клас містить методи: saveMemento, який використовується для збереження об’єкта memento; getMemento, який використовується для відновлення об’єкта запиту, та clearSavepoints, який використовується для очищення всіх точок збереження та видаляє всі збережені об’єкти memento.

Програмна реалізація

```
using System;
using System.Collections.Generic;
namespace Memento
{
 public class Program
 {
   public static void Main(string[] args)
   {
    CareTaker careTaker = new CareTaker();
    Originator originator = new Originator(5, 10, careTaker);
    Console.WriteLine("Початкові значення: " + originator + "\n");
    originator.X = originator.Y * 10;
    Console.WriteLine("Значення (x=y*10): " + originator + "\n");
    originator.createSavepoint("SAVE1");
    originator.Y = originator.X / 2;
    Console.WriteLine("Значення (y=x/2): " + originator + "\n");
    originator.undo();
    Console.WriteLine("Значення після undo: " + originator + "\n");
    originator.X = Math.Pow(originator.X, 2);
    Console.WriteLine("Значення (x=x*x): " + originator + "\n");
    originator.createSavepoint("SAVE2");
    Console.WriteLine("Значення: " + originator + "\n");
    originator.undo("SAVE1");
    Console.WriteLine("Отримання значень: " + originator + "\n");
    originator.undo("SAVE2");
    Console.WriteLine("Отримання значень: " + originator + "\n");
    originator.undoAll();
    Console.WriteLine("Значення після очищення всіх точок збереження: " + originator);

   Console.ReadLine();
  }
 }
 public class Originator
```

```
 {
  private double x;
  private double y;
  private string lastUndoSavepoint;
  internal CareTaker careTaker;
  public Originator(double x, double y, CareTaker careTaker)
  {
   this.x = x;
   this.y = y;
   this.careTaker = careTaker;
   createSavepoint("INITIAL");
  }
  public virtual double X
  {
   get { return x; }
   set { this.x = value; }
  }
  public virtual double Y
  {
   get { return y; }
   set { this.y = value; }
  }
  public virtual void createSavepoint(string savepointName)
  {
   careTaker.saveMemento(new Memento(this.x, this.y),
savepointName);
   lastUndoSavepoint = savepointName;
  }
  public virtual void undo()
  {
   OriginatorState = lastUndoSavepoint;
  }
  public virtual void undo(string savepointName)
  {
   OriginatorState = savepointName;
  }
  public virtual void undoAll()
  {
   OriginatorState = "INITIAL";
   careTaker.clearSavepoints();
  }
  private string OriginatorState
  {
   set
   {
    Memento mem = careTaker.getMemento(value);
    this.x = mem.X;
    this.y = mem.Y;
   }
  }
```

```
  public override string ToString()
  {
   return "X: " + x + ", Y: " + y;
  }
 }

 public class Memento
 {
  private double x;
  private double y;

  public Memento(double x, double y)
  {
   this.x = x;
   this.y = y;
  }
  public virtual double X {
   get { return x; }
  }
  public virtual double Y {
   get { return y; }
  }
 }

 public class CareTaker
 {
  private readonly IDictionary<string, Memento> savepointStorage =
new Dictionary<string, Memento>();

  public virtual void saveMemento(Memento memento, string
savepointName)
  {
   Console.WriteLine("Збереження значень..." + savepointName);
   savepointStorage[savepointName] = memento;
  }

  public virtual Memento getMemento(string savepointName)
  {
   Console.WriteLine("Повернення до точки збереження ..." +
savepointName);
   return savepointStorage[savepointName];
  }

  public virtual void clearSavepoints()
  {
   Console.WriteLine("Очищення всіх точок збереження...");
   savepointStorage.Clear();
  }
 }
}
```

4.5 Шаблон Стан (State)

Призначення

Стан – це поведінковий патерн проектування, який дозволяє об’єктам змінювати поведінку залежно від їхнього стану. Ззовні створюється враження, ніби змінився клас об’єкта. Стан дозволяє об’єкту змінювати свою поведінку, коли його внутрішній стан змінюється.

Стан об’єкта можна визначити як його точний стан у будь-який момент часу, залежно від значень його властивостей чи атрибутів. Набір методів, реалізованих класом, визначає поведінку його екземплярів. Щоразу, коли відбувається зміна значень його атрибутів, стан об’єкта змінюється.

Коли потрібно використовувати патерн

– Коли поведінка об’єкта залежить від його стану, і він має змінити свою поведінку під час виконання програми залежно від цього стану.

– Коли операції мають великі та розгалужені умовні оператори, які залежать від стану об’єкта.

Переваги

– Дозволяє позбавитись від великої кількості умовних операторів.

– Концентрує в одному місці код, пов’язаний з певним станом.

Недоліки

– Ускладнює код, якщо станів мало і вони рідко змінюються.

Загальна структура патерну

– Клас Context – визначає інтерфейс, що цікавить клієнтів. Зберігає екземпляр підкласу ConcreteState, який визначає поточний стан.

– State – визначає інтерфейс для інкапсуляції поведінки, пов’язаної з певним станом Context.

– ConcreteState – кожен підклас реалізує поведінку, пов’язану зі станом Context.

Умова. Задано акаунт для входу в певну систему.

Завдання. Реалізувати механізм доступу до контенту для заданого акаунту, використовуючи шаблон Стан.

UML-діаграму класів патерну Стан подано на рис. 4.5.

Приклад

Умова. Задано акаунт для входу в певну систему.

Завдання. Реалізувати механізм доступу до контенту для заданого акаунту, використовуючи шаблон Стан.

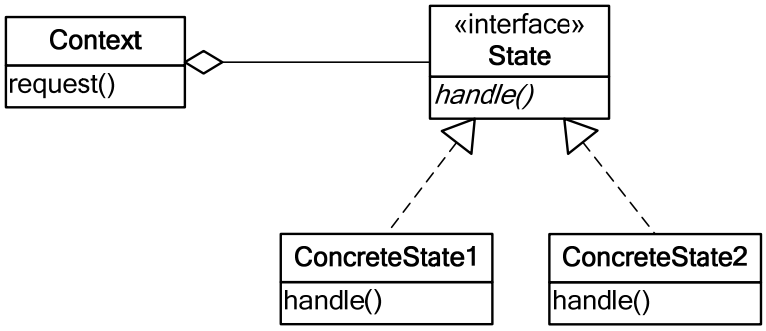


Рисунок 4.5 – UML-діаграма класів патерну Стан

Перелік учасників даного прикладу

– IState – інтерфейс, що визначає загальний інтерфейс доступу до контенту .

– Клас ActiveState – це конкретний стан, реалізує інтерфейс IState. Дозволяє перегляд контенту.

– Клас UnregistredState – це конкретний стан, реалізує інтерфейс IState. Перемикає незареєстрований акаунт на реєстрацію.

– Клас UnloggedState – це конкретний стан, реалізує інтерфейс IState. Повідомляє про перемикання незалогованого акаунту на вхід.

– Клас BannedState – це конкретний стан, реалізує інтерфейс IState. Повідомляє про блокування акаунту.

– Клас Account – це клас акаунту, який містить метод для перегляду контенту залежності від стану.

Програмна реалізація

```
using System;
using System.Text;
namespace State
{
 class Program
 {
  public static void Main(string[] args)
  {
   Console.OutputEncoding = Encoding.UTF8;
   Account account = new Account(new UnregistredState());
   account.ViewContent();
   account.State = new UnloggedState();
   account.ViewContent();
   account.State = new ActiveState();
   account.ViewContent();
   account.State = new BannedState();
   account.ViewContent();
   Console.Read();
  }
```

```
 }
 interface IState
 {
  void ViewContent();
 }
 class ActiveState : IState
 {
  public void ViewContent()
  {
   Console.WriteLine("Демонстрація контенту...");
  }
 }
 class UnregistredState : IState
 {
  public void ViewContent()
  {
   Console.WriteLine("Переведення на форму для реєстрації...");
  }
 }
 class UnloggedState : IState
 {
  public void ViewContent() {
  Console.WriteLine("Переведення на форму для входу...");
 }
 class BannedState : IState
 {
  public void ViewContent()
  {
   Console.WriteLine("Показ повідомлення про те що аккаунт користу-
вача заблокований...");
  }
 }
 class Account
 {
  public IState State { get; set; }
  public Account(IState state)
  {
   State = state;
  }
  public void ViewContent()
  {
   State.ViewContent();
  }
 }
}
```

4.6 Шаблон Шаблонний метод (Template Method)

Призначення

Шаблонний метод – це поведінковий патерн проектування, який визначає скелет алгоритму, передаючи деякі кроки до підкласів. Патерн дозволяє підкласам перевизначати певні кроки алгоритму без зміни його загальної структури.

Коли потрібно використовувати патерн

– Коли існує алгоритм, деякі кроки якого можна реалізувати кількома різними способами.

– Коли потрібно керувати розширеннями підкласів.

Переваги

– Дозволяє позбавитись від великої кількості умовних операторів.

– Концентрує в одному місці код, пов’язаний з певним станом.

Недоліки

– Ускладнює код, якщо станів мало і вони рідко змінюються.

Загальна структура патерну

– Клас AbstractClass – визначає абстрактні примітивні операції, які конкретні підкласи мають визначити для реалізації кроків алгоритму. Також реалізує шаблонний метод визначення скелета алгоритму. Шаблонний метод також викликає примітивні операції та операції, визначені в AbstractClass або в операціях інших об’єктів.

– Клас ConcreteClass – реалізує примітивні операції, які потрібно здійснити.

UML-діаграму класів патерну Шаблонний метод подано на рис. 4.6.

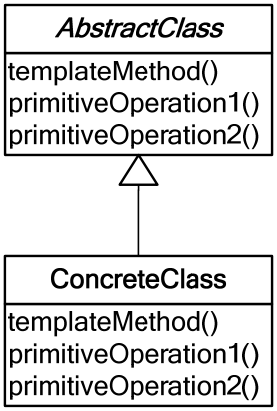


Рисунок 4.6 – UML-діаграма класів патерну Шаблонний метод

Приклад

Умова. Часто виникає задача, коли потрібно вставити деякі дані в різні бази даних. Наприклад, потрібно отримати деякі дані з CSV-файлу та вставити їх у базу даних MySQL. Інші дані надходять із текстового файлу, і їх потрібно вставити в базу даних Oracle. Єдина відмінність – драйвер і дані, решта кроків мають бути однаковими.

Завдання. Створити шаблоний клас, який використовується, щоб надати клієнтам шаблон для підключення та комунікації з різними базами даних.

Перелік учасників даного прикладу

– Клас ConnectionTemplate – абстрактний клас, який використовується для надання шаблону клієнтам для підключення та зв'язку з різними базами даних. Він надає кроки для підключення, зв'язку та закриття з'єднань. Всі ці кроки потрібні для виконання роботи. Клас забезпечує реалізацію за замовчуванням для деяких таких загальних кроків, як вставляння даних, закриття з'єднання, знищення об'єкта, які є однотипними для різних баз даних, та залишає конкретні кроки як абстрактні, що змушує клієнта надати їм реалізацію. Метод setDBDriver має бути реалізований користувачем для надання драйверів для конкретної бази даних. Метод setCredentials також залишається абстрактним, щоб користувач міг його реалізувати, оскільки вхідні дані можуть бути різними для різних баз даних. Метод run, який використовується для виконання кроків в певному порядку, визначений в цьому класі, оскільки кроки мають бути безпечними і не мають змінюватися жодним користувачем.

– Класи MySqLCSVCon та OracleTxtCon розширюють клас шаблону та забезпечують конкретну реалізацію деяких методів.

Програмна реалізація

```
using System;
namespace TemplateMethod
{
 public class Program
 {
  public static void Main(string[] args)
  {
   Console.WriteLine("For MYSQL....");
   ConnectionTemplate template = new MySqLCSVCon();
   template.run();
   Console.WriteLine("\nFor Oracle...");
   template = new OracleTxtCon();
   template.run();
   Console.ReadLine ( );
  }
 }
 public abstract class ConnectionTemplate
```

```
{
 private bool isLoggingEnable = true;
 public ConnectionTemplate()
 {
  isLoggingEnable = disableLogging();
 }
 public void run()
 {
  setDBDriver();
  logging("Drivers set [" + DateTime.Now + "]");
  setCredentials();
  logging("Credentails set [" + DateTime.Now + "]");
  connect();
  logging("Conencted");
  prepareStatement();
  logging("Statement prepared [" + DateTime.Now + "]");
  setData();
  logging("Data set [" + DateTime.Now + "]");
  insert();
  logging("Inserted [" + DateTime.Now + "]");
  close();
  logging("Conenctions closed [" + DateTime.Now + "]");
  destroy();
  logging("Object destoryed [" + DateTime.Now + "]");
 }
 public abstract void setDBDriver();
 public abstract void setCredentials();
 public virtual void connect()
 {
  Console.WriteLine("Setting connection...");
 }
 public virtual void prepareStatement()
 {
  Console.WriteLine("Preparing insert statement...");
 }
 public abstract void setData();
 public virtual void insert()
 {
  Console.WriteLine("Inserting data...");
 }
 public virtual void close()
 {
  Console.WriteLine("Closing connections...");
 }
 public virtual void destroy()
 {
  Console.WriteLine("Destroying connection objects...");
 }
 public virtual bool disableLogging()
 {
```

```
    return true;
   }
   private void logging(string msg)
   {
    if (isLoggingEnable)
     Console.WriteLine("Logging....: " + msg);
   }
  }
  public class MySqLCSVCon : ConnectionTemplate
  {
   public override void setDBDriver()
   {
    Console.WriteLine("Setting MySQL DB drivers...");
   }
   public override void setCredentials()
   {
    Console.WriteLine("Setting credentials for MySQL DB...");
   }
   public override void setData()
   {
    Console.WriteLine("Setting up data from csv file....");
   }
   public override bool disableLogging()
   {
    return false;
   }
  }
  public class OracleTxtCon : ConnectionTemplate
  {
   public override void setDBDriver()
   {
    Console.WriteLine("Setting Oracle DB drivers...");
   }
   public override void setCredentials()
   {
    Console.WriteLine("Setting credentials for Oracle DB...");
   }
   public override void setData()
   {
    Console.WriteLine("Setting up data from txt file....");
   }
  }
}
```

4.7 Шаблон Ланцюжок обов'язків (Chain of Responsibility)

Призначення

Ланцюжок обов'язків – це поведінковий патерн проектування, в якому група об'єктів поєднана в послідовності (ланцюжки), і відповідальність

(запит) надається з метою управління групою. Якщо об’єкт у групі може обробити конкретний запит, він робить це і повертає відповідну відповідь. В іншому випадку він пересилає запит наступному об’єкту в групі.

Таким чином, цей патерн забезпечує обробку об’єкта шляхом передачі його ланцюжком доти, доки не буде здійснена обробка однією з ланок.

Коли потрібно використовувати патерн

– Коли існує більше, ніж один об’єкт, який може обробити певний запит.

– Коли потрібно передати запит на виконання одному з декількох об’єктів, точно не визначаючи, якому саме об’єкту.

– Коли набір об’єктів задається динамічно.

– Коли принципово, щоб обробники виконувалися один за одним у визначеному порядку.

Переваги

– Дозволяє зменшити залежність між клієнтом та обробниками.

Недоліки

– Існує ймовірність того, що запит може залишитися ніким не опрацьованим.

Загальна структура патерну

– Handler – визначає інтерфейс для обробки запитів та використовується для передачі посилання наступному обробнику.

– ConcreteHandler **–** реалізує інтерфейс Handler**,** обробляє запити, за які відповідає. Може отримати доступ до свого наступника. Обробляє запит, якщо може; в іншому випадку передає запит наступному обробнику. Зберігає посилання на наступний об’єкт Handler**.**

– Клієнт **–** ініціює запит до об’єкта ConcreteHandler в ланцюжку.

Коли клієнт видає запит, запит поширюється по ланцюжку, поки об’єкт ConcreteHandler несе відповідальність за його обробку.

UML-діаграму класів патерну Ланцюжок обов’язків подано на рис. 4.7.

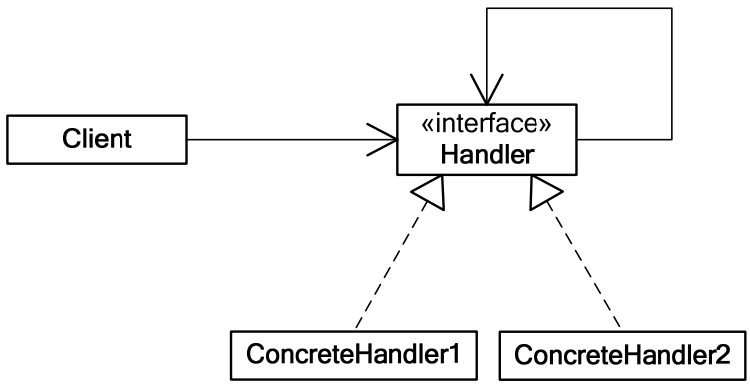


Рисунок 4.7 – UML-діаграма класів патерну Ланцюжок обов’язків

Приклад

Умова. При розробці програмного забезпечення часто виникає потреба у виведенні (логуванні) інформації для відстежування виконання програми. Таку інформацію про стан виконання програми або про помилки, що виникають при її виконанні, можна логувати різними способами: записувати в консоль, в файл або в базу даних.

Завдання. Створити певний логер, який забезпечить можливість логування повідомлень, виставлених в ланцюжок, вказаними способами з можливістю вибору способу логування інформації. Активність способів логування, вказати булевими змінними (прапорцями), що знаходяться в окремому класі.

Перелік учасників даного прикладу

– Клас LoggingMethod – клас, який містить булеві змінні (прапорці), що визначають активні елементи.

– Клас Logger – абстрактний клас, який містить поле такого ж типу, яке вказує на наступний логер, тобто обробник в ланцюжку. Є метод Logg, в який передаються LoggingMethod та саме повідомлення

– Класи ConsoleLogger, FileLogger та DatabaseLogger забезпечують конкретну реалізацію методу Logg. Вони перевіряють вибір методу та відповідно до нього виводять повідомлення. Також відбувається перевірка наявності наступного обробника і у випадку його відсутності метод не буде виконуватись.

Програмна реалізація

```
using System;
namespace ChainOfResponsibility
{
 class Program
 {
  public static void Main(string[] args)
  {
   LoggingMethod loggingMethod = new LoggingMethod
   {
    Console = true, File = false, Database = true
   };
   Logger consoleLogger = new ConsoleLogger();
   Logger fileLogger = new FileLogger();
   Logger databaseLogger = new DatabaseLogger();
   consoleLogger.NextLogger = fileLogger;
   fileLogger.NextLogger = databaseLogger;
```

```
    consoleLogger.Log(loggingMethod, "Fatal Error");
   }
}
class LoggingMethod
{
 public bool Console { get; set; }
 public bool File { get; set; }
 public bool Database { get; set; }
}
abstract class Logger
{
 public Logger NextLogger { get; set; }
  public abstract void Log(LoggingMethod loggingMethod, string
message);
 }
 class ConsoleLogger : Logger
 {
  public override void Log(LoggingMethod loggingMethod, string
message)
  {
   if (loggingMethod.Console)
   Console.WriteLine($"Console: {message}");
   NextLogger?.Log(loggingMethod, message);
  }
 }
 class FileLogger : Logger
 {
  public override void Log(LoggingMethod loggingMethod, string
message)
  {
   if (loggingMethod.File)
    Console.WriteLine($"File: {message}");
   NextLogger?.Log(loggingMethod, message);
  }
 }
 class DatabaseLogger : Logger
 {
  public override void Log(LoggingMethod loggingMethod, string
message)
  {
   if (loggingMethod.Database)
    Console.WriteLine($"Database: {message}");
   NextLogger?.Log(loggingMethod, message);
  }
 }
}
```

4.8 Шаблон Посередник (Mediator)

Призначення

Посередник – це поведінковий патерн проектування, який дозволяє зменшити зв'язаність великої кількості класів між собою шляхом переміщення цих зв'язків до одного класу, який є посередником.

Таким чином, цей патерн покращує зв'язність шляхом утримання об'єктів від прямих посилань один на одного та дозволяє незалежно змінювати взаємодію.

Коли потрібно використовувати патерн

– Коли складно змінювати деякі класи, оскільки вони мають велику кількість неструктурованих зв'язків з іншими класами, які важко зрозуміти.

– Коли повторне використання об'єкта є складним, оскільки він посилається на багато інших об'єктів і взаємодіє з ними.

Переваги

– Дозволяє уникнути залежності між компонентами.

– Дозволяє повторно використовувати компоненти.

– Дозволяє централізувати керування зв'язками між об'єктами.

Недоліки

– Існує ймовірність того, що клас Посередника може стати занадто великим.

Загальна структура патерну

– Mediator – визначає інтерфейс для спілкування з об'єктами Colleague.

– ConcreteMediator – реалізує спільну поведінку, координуючи об'єкти Colleague.

– Colleague – визначає інтерфейс для взаємодії з об'єктом Mediator

– ConcreteColleague – кожен клас знає свій об'єкт Mediator. Кожен клас Colleague спілкується зі своїм посередником.

UML-діаграму класів патерну Посередник подано на рис. 4.8.

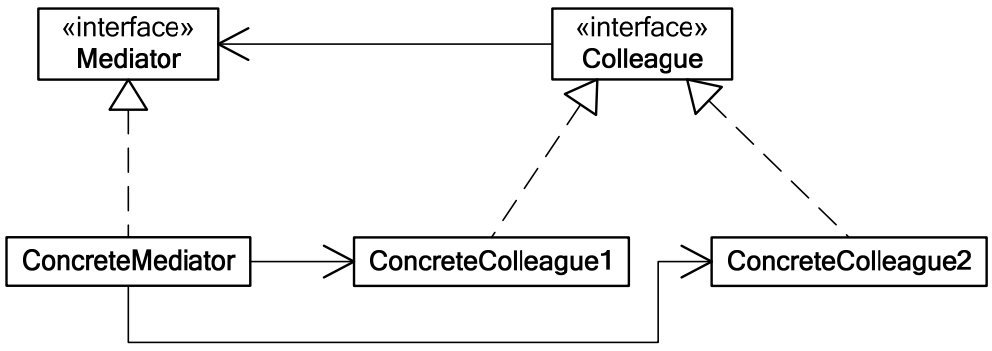


Рисунок 4.8 – UML-діаграма класів патерну Посередник

Приклад

Умова. Робота автоматизованої пральної машини полягає у виконанні певної послідовності програм при натисканні користувачем на відповідну кнопку. Зокрема, потрібно відкрити воду, наповнити машину водою, нагріти її до певної визначеної температури, запустити двигун та запустити центрифугу на визначеній кількості обертів.

Завдання. Розробити програмне забезпечення для роботи пральної машини, використовуючи шаблон Посередник.

Перелік учасників даного прикладу

– Інтерфейс MachineMediator – це інтерфейс, який діє як загальний посередник. Інтерфейс містить методи виклику одного об’єкта іншим.

– Клас CottonMediator реалізує інтерфейс MachineMediator і дає потрібні методи, які виконуються об’єктами-колегами, щоб виконати роботу. Клас-посередник просто викликає метод об’єкта-колеги від імені іншого об’єкта-колеги, щоб досягти цього.

– Інтерфейс Colleague – визначає інтерфейс колеги.

– Клас Button є класом колеги, який містить посилання на посередника. Користувач натискає кнопку, яка викликає метод press() цього класу, який, у свою чергу, викликає метод start() конкретного класу посередника. Цей метод start()викликає метод start() машинного класу від імені класу Button.

– Класи Machine, Valve, Heater також є класами колег, які містять посилання на посередника та методи, потрібні для вирішення поставленої задачі, аналогічні методу start() в попередньому класі.

– Класи Sensor та Motor – допоміжні класи. Клас Sensor використовується класом Heater для перевірки температури. Клас Motor – використовується в CottonMediator для запуску двигуна.

Програмна реалізація

```
using System;
namespace Mediator
{
 public class Program
 {
  public static void Main(string[] args)
  {
   MachineMediator mediator = null;
   Sensor sensor = new Sensor();
   Motor motor = new Motor();
   Machine machine = new Machine();
   Heater heater = new Heater();
   Valve valve = new Valve();
   Button button = new Button();
   mediator = new CottonMediator(machine, heater, motor, sensor,
valve);
```

```
 button.Mediator = mediator;
 machine.Mediator = mediator;
 heater.Mediator = mediator;
 valve.Mediator = mediator;
 button.press();
}
public interface MachineMediator
{
 void start();
 void wash();
 void open();
 void closed();
 void on();
 void off();
 bool checkTemp(int temp);
}
public interface Colleague
{
 MachineMediator Mediator { set; }
}
public class Button : Colleague
{
 private MachineMediator mediator;
 public MachineMediator Mediator
 {
  set { this.mediator = value; }
 }
 public virtual void press()
 {
  Console.WriteLine("Кнопку натиснуто");
  mediator.start();
 }
}
public class Machine : Colleague
{
 private MachineMediator mediator;
 public MachineMediator Mediator
`{
  set { this.mediator = value; }
 }
 public virtual void start()
 {
  mediator.open();
 }
 public virtual void wash()
 {
  mediator.wash();
 }
}
public class Heater : Colleague
```

```
{
 private MachineMediator mediator;
 public MachineMediator Mediator
 {
  set { this.mediator = value; }
 }
 public virtual void on(int temp)
 {
  Console.WriteLine("Тен включився");
  if (mediator.checkTemp(temp))
  {
   Console.WriteLine("Температуру встановлено на " + temp);
   mediator.off();
  }
 }
 public virtual void off()
 {
  Console.WriteLine("Тен виключився");
  mediator.wash();
 }
}
public class Valve : Colleague
{
 private MachineMediator mediator;
 public MachineMediator Mediator
 {
  set { this.mediator = value; }
 }
 public virtual void open()
 {
  Console.WriteLine("Клапан відкрився");
  Console.WriteLine("Наповнення водою");
  mediator.closed();
 }
 public virtual void closed()
 {
  Console.WriteLine("Клапан закрився");
  mediator.on();
 }
}
public class CottonMediator : MachineMediator
{
 private readonly Machine machine;
 private readonly Heater heater;
 private readonly Motor motor;
 private readonly Sensor sensor;
 private readonly Valve valve;
 public CottonMediator(Machine machine, Heater heater, Motor
```

```
motor, Sensor sensor, Valve valve)
    {
     this.machine = machine;
     this.heater = heater;
     this.motor = motor;
     this.sensor = sensor;
     this.valve = valve;
     Console.WriteLine("COTTON program");
    }
    public void start()
    {
     machine.start();
    }
    public void wash()
    {
     motor.startMotor();
     motor.rotateDrum(800);
    }
    public void open()
    {
     valve.open();
    }
    public void closed()
    {
     valve.closed();
    }
    public void on()
    {
     heater.on(60);
    }
    public void off()
    {
     heater.off();
    }
    public bool checkTemp(int temp)
    {
     return sensor.checkTemperature(temp);
    }
   }
   public class Sensor
   {
    public virtual bool checkTemperature(int temp)
    {
     Console.WriteLine("Температура досягла " + temp + " град.");
     return true;
    }
   }
   public class Motor
   {
    public virtual void startMotor()
```

```
    {
     Console.WriteLine("Запуск двигуна");
    }
    public virtual void rotateDrum(int rpm)
    {
     Console.WriteLine("Віджим при " + rpm + " об.");
    }
   }
  }
}
```

4.9 Шаблон Відвідувач (Visitor)

Призначення

Відвідувач – це поведінковий патерн проектування, який дозволяє додавати до програми нові операції, не змінюючи класи об'єктів, над якими ці операції можуть виконуватися.

Щоб досягти цього, шаблон Відвідувач пропонує визначити операцію в окремому класі, який називається Visitor. Це відокремлює операцію від колекції об'єктів, з якою вона оперує. Для кожної нової операції, яку потрібно визначити, створюється новий клас відвідувача. Оскільки операція має виконуватися над набором об'єктів, відвідувачу потрібен спосіб доступу до загальнодоступних членів цих об'єктів.

Коли потрібно використовувати патерн

– Коли структура об'єкта містить багато класів об'єктів з різними інтерфейсами, і потрібно виконувати операції над цими об'єктами, які залежать від їх конкретних класів.

– Коли з об'єктами в структурі потрібно виконувати багато неспоріднених операцій, і уникати «засмічення» класів цими операціями.

– Коли класи, що визначають структуру об'єкта, рідко змінюються, але часто потрібно визначити нові операції над структурою.

Переваги

– Дозволяє об'єднати споріднені методи в одному класі.

– Дозволяє спростити додавання методів, які працюють зі складними структурами об'єктів.

Недоліки

– Існує ймовірність порушення інкапсуляції елементів.

– Неефективний при частозмінюваній ієрархії елементів.

Загальна структура патерну

– Інтерфейс Visitor – оголошує метод відвідування для кожного класу ConcreteElement в об'єктній структурі. Ім'я та сигнатура методу ідентифікують клас, який надсилає відвідувачу запит на відвідування. Це дозволяє відвідувачу визначити конкретний клас елемента, який відвідується. Тоді

відвідувач може отримати доступ до елемента безпосередньо через його конкретний інтерфейс.

– Клас ConcreteVisitor – реалізує методи, оголошені Visitor. Кожна операція реалізує фрагмент алгоритму, визначеного для відповідного класу об’єкта в структурі. ConcreteVisitor надає контекст алгоритму та зберігає його локальний стан. Цей стан часто накопичує результати під час обходу структури.

– Інтерфейс Element – визначає метод accept, який приймає відвідувача як аргумент.

– Клас ConcreteElement – реалізує метод accept.

– Клас ObjectStructure – може перерахувати елементи. Може надавати інтерфейс високого рівня, щоб відвідувач міг відвідувати елементи. Може бути композитом або колекцією (списком або набором).

UML-діаграму класів патерну Відвідувач наведено на рис. 4.9.

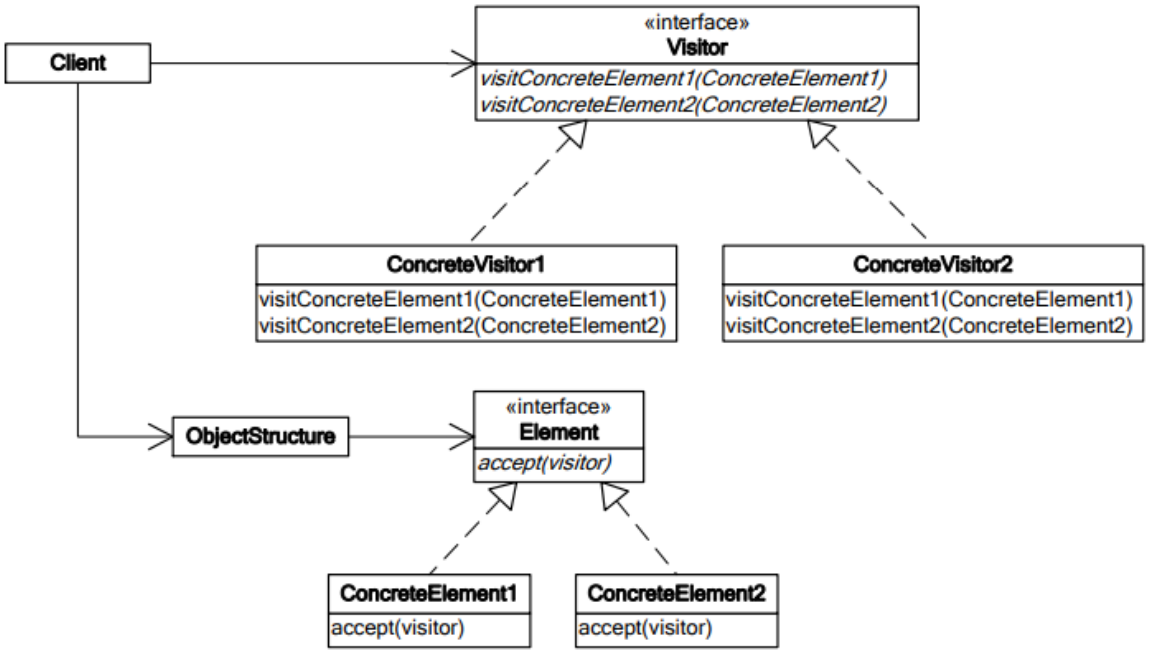


Рисунок 4.9 – UML-діаграма класів патерну Відвідувач

Приклад

Умова. За основу для даного прикладу візьмемо приклад, наведений для шаблону Композіт.

Завдання. Додати клас css до тегів html з вказаного прикладу, без зміни класів елементів. Використати шаблон Відвідувач.

Перелік учасників даного прикладу

В цьому прикладі не буде наводитись повний текст програми, а тільки зміни, які внесено до коду в прикладі до шаблону Композит.

– Інтерфейс Element – містить метод accept з аргументом типу Visitor. Цей інтерфейс буде реалізовано всіма класами, які мають дозволити відві-

дувачам відвідувати їх. В даному випадку клас HtmlTag реалізує інтерфейс Element, оскільки HtmlTag є батьківським абстрактним класом для всіх конкретних класів html, конкретні класи успадкують і замінять метод Accept інтерфейсу Element.

Відмітимо, що в клас HtmlTag додано два методи – getStartTag і getEndTag на відміну від прикладу, показаного в шаблоні Композит.

– Інтерфейс Visitor – містить методи відвідування з аргументом класу, який реалізує інтерфейс Element.

– Класи CssClassVisitor, StyleVisitor – конкретні класи відвідувачів. Перший додає відвідувача класу css до всіх тегів html, а другий змінює ширину тегу за допомогою атрибута style тегу html.

Програмна реалізація

```
 public interface Element
 {
  void accept(Visitor visitor);
 }
 public interface Visitor
 {
  void visit(HtmlElement element);
  void visit(HtmlParentElement parentElement);
 }
 public class CssClassVisitor : Visitor
 {
  public void visit(HtmlElement element)
  {
   element.StartTag = element.StartTag.Replace(">", "
class='visitor'>");
  }
  public void visit(HtmlParentElement parentElement)
  {
   parentElement.StartTag = parentElement.StartTag.Replace(">", "
class='visitor'>");
  }
 }
 public class StyleVisitor : Visitor
 {
  public void visit(HtmlElement element)
  {
   element.StartTag = element.StartTag.Replace(">", "
style='width:46px;'>");
  }
  public void visit(HtmlParentElement parentElement)
  {
   parentElement.StartTag = parentElement.StartTag.Replace(">", "
style='width:58px;'>");
  }
 }
```

В клас HtmlTag потрібно додати метод:

```
public abstract void accept( Visitor visitor );
```

В класи HtmlParentElement та HtmlElement потрібно додати метод:

```
public override void accept(Visitor visitor)
  {
   visitor.visit(this);
  }
```

Відповідні зміни потрібно внести і в клас клієнта:

```
Visitor cssClass = new CssClassVisitor();
Visitor style = new StyleVisitor();
parentTag = new HtmlParentElement("<html>");
parentTag.StartTag = "<html>";
parentTag.EndTag = "</html>";
parentTag.accept(style);
parentTag.accept(cssClass);
```

І аналогічно для кожного html-елементу.

4.10 Шаблон Команда (Command)

Призначення

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

Коли потрібно використовувати патерн

– Коли потрібно параметризувати об’єкти за допомогою дії, яку потрібно виконати.

– Коли потрібно вказувати, ставити в чергу та виконувати запити в різний час.

– Коли потрібна підтримка операції скасування.

– Коли потрібна підтримка журналу змін, щоб їх можна було повторно застосувати в випадку збою системи.

Переваги

– Дозволяє уникнути прямої залежності між об’єктами, які викликають операції, та об’єктами, які їх безпосередньо виконують.

– Дозволяє реалізувати скасування та повторення операцій.

– Дозволяє реалізувати відкладений запуск операцій.

Недоліки

– Може ускладнювати код програми.

Загальна структура патерну

– Command – оголошує інтерфейс для виконання операції.

– ConcreteCommand – визначає прив'язку між об'єктом Receiver та дією. Реалізує метод Execute шляхом виклику відповідної операції з Receiver.

– Client – створює об'єкт ConcreteCommand та призначає його об'єкту Receiver.

– Invoker – просить команду виконати запит.

– Receiver – знає, як виконувати операції, пов'язані з виконанням запиту.

UML-діаграму класів патерну Команда наведено на рис. 4.11.

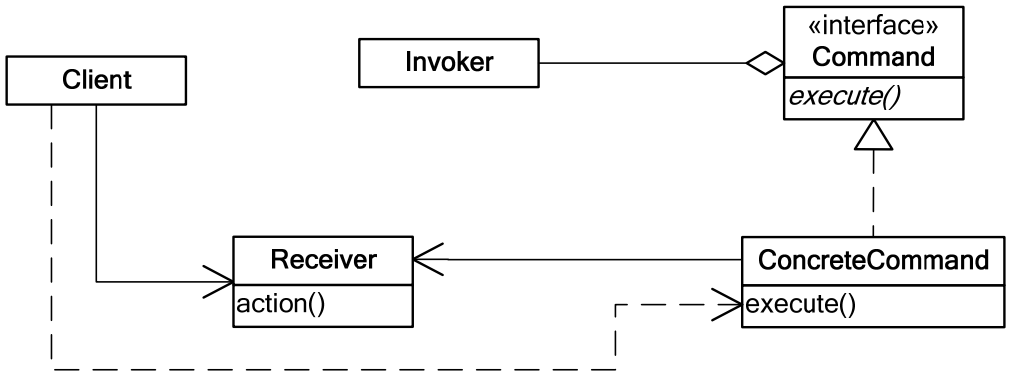


Рисунок 4.11 – UML-діаграма класів патерну Команда

Приклад

Умова. Задано елемент, який має виконувати збереження даних. Як такий елемент можна використати кнопку, елемент контекстного меню або комбінацію клавіш тощо

Завдання. Створити обробник такого елемента за допомогою шаблону Команда.

Перелік учасників даного прикладу

– Інтерфейс ICommand – визначає загальний інтерфейс для команди. Містить метод Execute().

– Клас SaveCommand – клас, конкретна команда, яка буде зберігати певні дані. Викликає певний метод класу Receiver.

– Клас Receiver – клас, який містить логіку збереження даних.

– Клас Invoker – клас, який містить посилання на Команду. Таких класів може бути декілька (наприклад, для кнопки, для контекстного меню, для комбінації клавіш).

Програмна реалізація

```
using System;
namespace Comand_Sample
{
 class Program
 {
  static void Main(string[] args)
  {
   Invoker invoker = new Invoker();
   Receiver receiver = new Receiver();
   invoker.SetCommand(new SaveCommand(receiver));
   invoker.ExecuteCommand();
   Console.ReadKey();
  }
 }
 public interface ICommand
 {
  void Execute();
 }
 class SaveCommand : ICommand
 {
  private Receiver _receiver;
  public SaveCommand(Receiver receiver)
  {
   this._receiver = receiver;
  }
  public void Execute()
  {
   Console.WriteLine("SaveCommand: transfering direction to
receiver.");
   this._receiver.Save();
  }
 }
 class Receiver
 {
  public void Save()
  {
   // Логіка збереження даних
   Console.WriteLine("Receiver: Save complete.");
  }
 }
 class Invoker
 {
  private ICommand _command;
  public void SetCommand(ICommand command)
  {
   this._command = command;
  }
```

```
  public void ExecuteCommand()
  {
   Console.WriteLine("Invoker: Process is in progress...");
   if (this._command is ICommand)
    this._command.Execute();
    // Тут можливі додаткові дії, які супроводжують кожну команду
    // "Відправник" може мати свою логіку
   Console.WriteLine("Invoker: Process is complete.");
  }
 }
}
```

4.11 Шаблон Інтерпретатор (Interpreter)

Призначення

Інтерпретатор – це поведінковий патерн проектування, який визначає подання граматичних правил певної мови, а також інтерпретатор, який використовує це подання для інтерпретації речень у мові.

В загальному, мови складаються з набору граматичних правил. Дотримуючись цих граматичних правил, можна побудувати різні речення. Іноді програмі може знадобитися обробляти повторення подібних запитів, які є комбінацією набору граматичних правил. Ці запити відрізняються, але схожі в тому сенсі, що всі вони складаються з використанням одного набору правил.

Простим прикладом цього може бути набір різних арифметичних виразів, приєднаних до програми калькулятора. Хоча кожен такий вираз відрізняється, всі вони побудовані з використанням основних правил, які складають граматику мови арифметичних виразів.

Коли потрібно використовувати патерн

– Коли граматика мови є простою.

– Коли ефективність не є критичною проблемою.

Переваги

– Легко змінювати та розширювати граматику.

– Простота реалізації граматики.

Недоліки

– Дуже рідко використовується на практиці.

– Проблеми супроводження складних граматик.

Загальна структура патерну

– ІнтерфейсAbstractExpression – оголошує абстрактний метод інтерпретації interpret(), який є спільним для всіх вузлів абстрактного синтаксичного дерева.

– Клас TerminalExpression – реалізує метод інтерпретації, пов’язаний з термінальними символами в граматиці. Для кожного символу граматики створюється об’єкт TerminalExpression.

– Клас NonTerminalExpression – презентує правило граматики. Для кожного окремого правила граматики створюється свій об’єкт NonTerminalExpression.

– Клас Content – містить загальну інформацію для інтерпретатора. Може використовуватися об’єктами термінальних та нетермінальних виразів для збереження стану операцій та подальшого доступу до збереженого стану.

– Клас Client – створює абстрактне синтаксичне дерево, вузлами якого є об’єкти TerminalExpression та NonTerminalExpression. Викликає метод інтерпретації.

UML-діаграму класів патерну Інтерпретатор наведено на рис. 4.11.

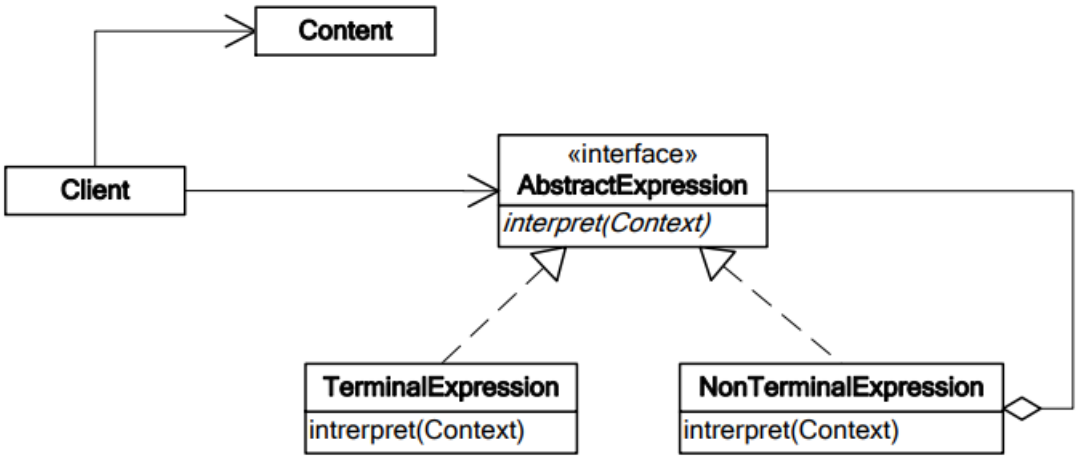


Рисунок 4.11 – UML-діаграма класів патерну Інтерпретатор

Питання для самоперевірки

1. Коротко охарактеризуйте шаблони поведінки.
2. Наведіть переваги та недоліки використання шаблону Стратегія.
3. Наведіть UML-діаграму для шаблону Стан.
4. Охарактеризуйте які можливості дає використання шаблону Знімок. Наведіть приклад з реального життя.
5. Поясніть в яких випадках використовується патерн Посередник. Наведіть його переваги та недоліки.
6. Коли доцільно використовувати шаблон Ітератор?
7. Охарактеризуйте шаблон Ланцюжок обов’язків. Наведіть приклад, пов’язаний з реальним життям.

8. Поясніть переваги використання шаблону Відвідувач.
9. Наведіть UML-діаграму для шаблону Стратегія.
10. Охарактеризуйте шаблон Шаблоннний метод.
11. Поясніть в яких випадках використовується патерн Команда. Наведіть його переваги та недоліки.
12. Наведіть UML-діаграму для шаблону Спостерігач.
13. Наведіть UML-діаграму для шаблону Інтерпретатор.
14. Поясніть призначення класів, що використовуються для шаблону Спостерігач.
15. Наведіть UML-діаграму для шаблону Знімок.

ЛІТЕРАТУРА

1. Fowler M. UML. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Third Edition. Boston : Addison-Wesle, 2003. 208 p.

2. Rumbaugh J., Jacobson I., Booch G. The Unified Modeling Language Reference Manual. 2nd Edition. Boston : Addison-Wesle, 2010. 721 p.

3. Design patterns. Elements of reusable object-oriented software / Gamma E., Helm R., Jonson R., Vlissides J. Boston : Addison-Wesle, 1995. 395 p.

4. Влиссидес Дж. Применение шаблонов проектирования. Дополнительные штрихи ; пер. с англ. М. : Издательский дом «Вильямс», 2003. 144 с.

5. Фрімен Ерік, Робсон Елізабет. «Head First. Патерни проєктування». Харків : Фабула. 2020. 672 с.

6. Андрій Будай. Дизайн-патерни – просто, як двері. Львів : 2012. 90 с. Режим доступу : https://sites.google.com/site/designpatternseasy/

7. Паттерны проектирования в C# и .NET. Режим доступу : https://metanit.com/sharp/patterns/

8. Патерни проектування. Режим доступу : https://refactoring.guru/uk/design-patterns.

9. Design Patterns. Режим доступу : https://sourcemaking.com/design_patterns.

Електронне навчальне видання

Войцеховська Олена Валеріївна
Черняк Олександр Іванович

ШАБЛОНИ ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Навчальний посібник

Рукопис оформила *О. Войцеховська*

Редактор *В. Дружиніна*

Оригінал-макет підготовано в *РВВ ВНТУ*

Підписано до видання 27.07.2022 р.
Гарнітура Times New Roman.
Зам. № Р2022-058.

Видавець та виготовлювач
Вінницький національний технічний університет,
редакційно-видавничий відділ.
ВНТУ, ГНК, к. 114.
Хмельницьке шосе, 95,
м. Вінниця, 21021.
Тел. (0432) 65-18-06.
press.vntu.edu.ua;
Email: irvc.vntu@gmail.com
Свідоцтво суб'єкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.