Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №5
Технології розроблення програмного забезпечення
«Патерни проєктування»

Виконав
студент групи ІА–34:
Бородай А.С.

Київ 2025

Зміст

Теоретичні відомості

Шаблон «Adapter»

Призначення патерну «Adapter»: використовується для адаптації інтерфейсу одного об'єкту до іншого. Наприклад, існує декілька бібліотек для роботи з принтерами, проте кожна має різний інтерфейс. Має сенс розробити уніфікований інтерфейс, і реалізувати відповідні адаптери для приведення бібліотек до уніфікованого інтерфейсу. Це дозволить в програмі звертатися до загального інтерфейсу, а не приводити різні сценарії роботи залежно від способу реалізації бібліотеки. Адаптери також називаються "wrappers" (обгортками).

Проблема:

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

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

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

Рішення:

Для вирішення цих проблем можна використати патерн Адаптер. Визначаємо загальний інтерфейс IPlayer для програвання музики через компоненти. Далі для кожного компонента робимо свій адаптер. Адаптер має посилання на об'єкт компонента та реалізує інтерфейс викликаючи методи з компонента, який він адаптує. Класи, які будуть працювати з адаптером через цей інтерфейс не будуть знати інтерфейси кожного специфічного компонента.

При такій реалізації, якщо буде потрібно добавити підтримку нового формату, то просто створюється новий адаптер, який обгортає новий компонент, але робота з цим

адаптером буде виконуватися через існуючий інтерфейс. При цьому існуючий код не буде зазнавати змін, а значить і не "поламаємо" те що вже працює.

Переваги та недоліки:

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

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

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

Шаблон «Builder»

Призначення «Builder»: використовується для відділення процесу створення об'єкту від його представлення. Це доречно у випадках, коли об'єкт має складний процес створення або коли об'єкт повинен мати декілька різних форм створення (наприклад, при конвертації тексту з формату у формат).

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

Рішення: Кожен з цих етапів може бути абстрагований в окремий метод будівельника.

Це дасть наступні вигоди:

- Гнучкіший контроль над процесом створення сторінки;

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

Переваги та недоліки:

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

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

Шаблон «Command»

Призначення «Command»: перетворення звичайного виклику методу в клас. Таким чином дії в системі стають повноправними об'єктами. Це зручно в наступних випадках:

• Коли потрібна розвинена система команд – відомо, що команди будуть добавлятися;

• Коли потрібна гнучка система команд – коли з'являється необхідність додавати командам можливість відміни, логування і інш.;

• Коли потрібна можливість складання команд або виклику команд в певний час.

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

Проблема: Ви реалізуєте товстий клієнт який має багатий візуальний інтерфейс: має меню, кнопки і контекстне меню. Кожна дія, яку можна виконати, має три варіанти виконання – через меню, натисканням кнопки та через контекстне меню. Реалізацію кожної дії можна розмістити в обробнику візуального елементу, але тоді потрібно буде продублювати цей функціонал для меню, кнопки та контекстного меню. Таким чином ми будемо мати дублювання коду і при зміні функціоналу змінювати його в трьох місцях. Додатковим викликом буде реалізувати автоматизоване тестування такої системи, тому що нам необхідно емулювати натискання кнопок користувачем.

Рішення: Виділення функціоналу який виконується по натисканню на кнопку в окремий клас дозволяє відвязати візуальну частину від логіки обробки. Таким чином ми будемо мати шар з UI елементами і шар з логікою обробки дій виконаних на UI. Це дозволяє один і той самий об'єкт з шару логіки обробки дій використати для реакцію на

натискання кнопки і пункту меню і контекстного меню. Кнопки тепер не знають про конкретні класи реалізації команд, а взаємодіють з об'єктами команд через загальний інтерфейс. Ще одна перевага, що тепер ми можемо достатньо просто реалізувати механізм enable-disable для кнопок та контекстного меню через виклик у об'єкта команди метода IsEnabled() або через прив'язку (binding) на поле IsEnabled у об'єкта команди. При такій реалізації також набагато простіше організувати тестування застосунку, тому що ми тепер не потрібно емулювати дії користувача, а достатньо протестувати шар логіки обробки використовуючи модульні тести (unit tests).

Переваги та недоліки:

+ Ініціатор виконання команди не знає деталей реалізації виконавця команди.

+ Підтримує операції скасування та повторення команд.

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

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

Шаблон «Chain of Responsibility»

Призначення «Chain of Responsibility»: коли підписання відповідного документу проходить від його складання у одного із співробітників компанії через менеджера і начальника до головного начальника, який ставить свій підпис.

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

Рішення: Основна ідея в тому, що нам не потрібно мати загальний метод формування контекстного меню. Ми можемо зробити загальний інтерфейс з методом UpdateContextMenu() і реалізувати цей метод в усіх візуальних компонентах. Додатково для візуальних компонентів добавляємо поле для переходу на "батьківський" елемент, якій в собі містить поточний візуальний компонент, а в реалізації UpdateContextMenu в кінці додаємо виклик цього метода у "батьківського" елемента. Якщо елемент не потребує додавання пунктів в контекстне меню, він просто викликає цей метод у наступного елемента в ланцюжку.

Переваги та недоліки:

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

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

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

Шаблон «Prototype»

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

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

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

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

Переваги та недоліки:

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

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

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

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

Тема: Патерни проектування.

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

Завдання:

• Ознайомитись з короткими теоретичними відомостями.

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

• Реалізувати один з розглянутих шаблонів за обраною темою.

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

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

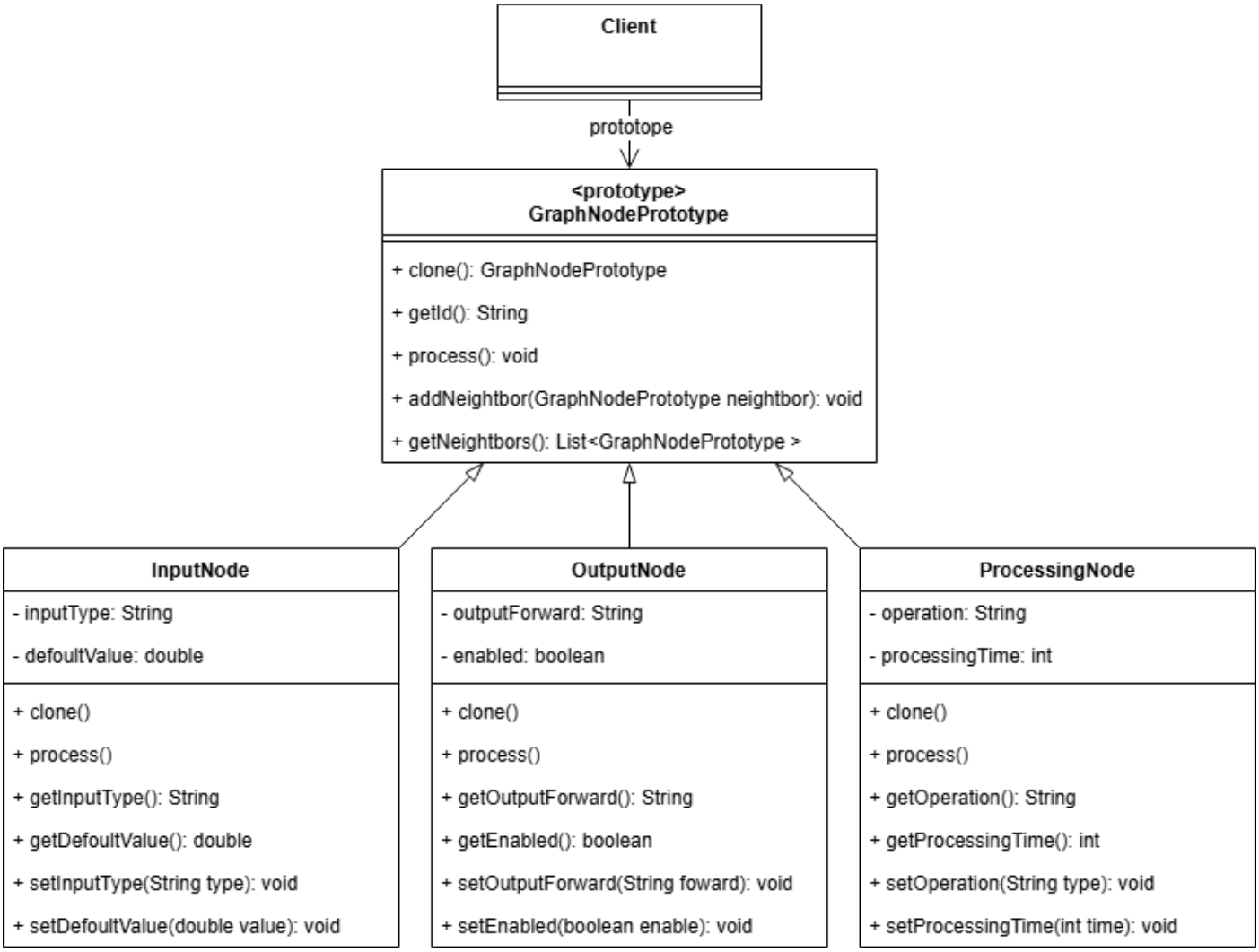
Хід роботи

20. **Mind-mapping software**

(strategy, prototype, abstract factory, bridge, composite, SOA)

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

ClassDiagram (Prototype)



Висновок

Під час виконання лабораторної роботи я ознайомився із шаблонами «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype».

Мною були розроблені основні класи, які реалізують шаблон Prototype та її поведінку. Конкретно цей шаблон, аналізуючи предметну область та вимоги до системи, не зовсім доцільно використовувати у конкретній реалізації проеєкту. Проте це не завадило мені побудувати обраний шаблон.

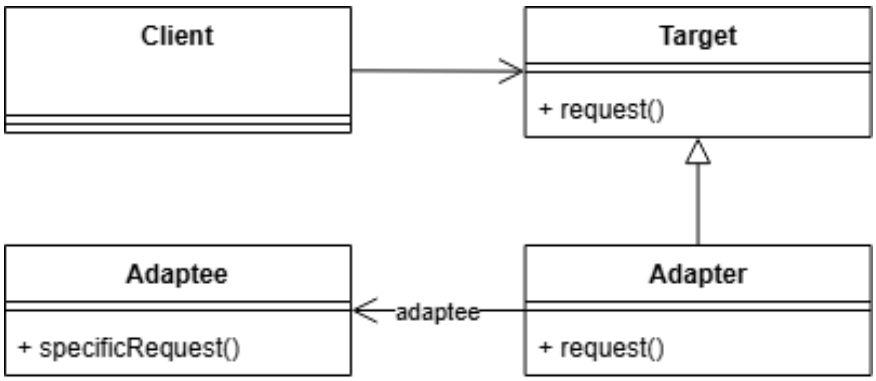
Нові шаблони будуть мною використані для подальшої реалізації системи та подальшого її розуміння.

Відповіді на питання для самоперевірки

1. Яке призначення шаблону «Адаптер»?

Це перетворення інтерфейсу одного класу на інтерфейс, очікуваний клієнтом. Адаптер дозволяє класам із несумісними інтерфейсами працювати разом, виступаючи своєрідним «перехідником».

2. Нарисуйте структуру шаблону «Адаптер».



3. Які класи входять в шаблон «Адаптер», та яка між ними взаємодія?

Client (Клієнт) – клас, який використовує об'єкти з цільовим інтерфейсом (Target).

Target (Ціль) – інтерфейс (або абстрактний клас), який очікує побачити Client.

Adaptee (Адаптований) – існуючий клас з несумісним інтерфейсом, який потрібно адаптувати.

Adapter (Адаптер) – конкретний клас, який реалізує інтерфейс Target і агрегує (або успадковує, у випадку класового адаптера) об'єкт Adaptee.

Client викликає метод Request() у об'єкта, який він сприймає як Target. Цим об'єктом насправді є екземпляр Adapter. Adapter отримує цей виклик і делегує його методу SpecificRequest() об'єкта Adaptee, який він зберігає. Результат роботи Adaptee повертається через Adapter до Client.

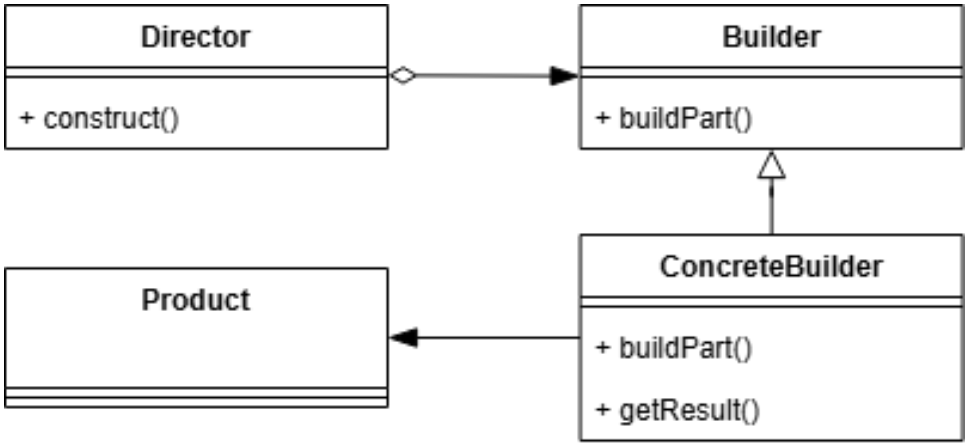
4. Яка різниця між реалізацією «Адаптера» на рівні об’єктів та на рівні класів?

Адаптер на рівні об'єктів дикористовує композицію, а отже містить посилання на об'єкт Adaptee і делегує йому роботу. Адаптер на рівні класів використовує множинне успадкування, тому адаптер успадковується від Target та від Adaptee. Таким чином, він отримує функціональність Adaptee без необхідності створювати окремий об'єкт.

5. Яке призначення шаблону «Будівельник»?

Відділити конструювання складного об'єкта від його представлення, щоб один і той же процес конструювання міг створювати різні представлення.

6. Нарисуйте структуру шаблону «Будівельник».



7. Які класи входять в шаблон «Будівельник», та яка між ними взаємодія?

Director (Керівник) – відповідає за виконання послідовності кроків для побудови продукту. Він використовує інтерфейс Builder і не залежить від конкретних класів будівельників або продуктів.

Builder (Будівельник) – абстрактний інтерфейс, який визначає кроки для створення частин продукту.

ConcreteBuilder (Конкретний будівельник) – реалізує інтерфейс Builder і надає конкретну реалізацію кроків побудови. Він також відповідає за створення та зберігання створюваного продукту (Product) і надає метод для його отримання (GetResult).

Product (Продукт) – складний об'єкт, що створюється.

Client створює об'єкт ConcreteBuilder і передає його об'єкту Director. Director викликає методи BuildPart() у будівельника, керуючи процесом побудови. Після завершення процесу, Client отримує готовий продукт через метод GetResult() у ConcreteBuilder.

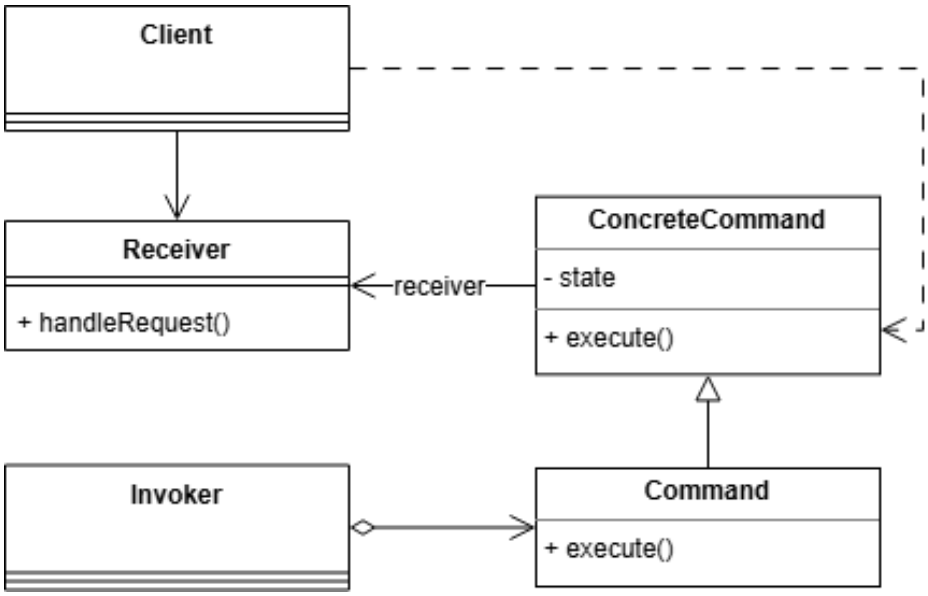
8. У яких випадках варто застосовувати шаблон «Будівельник»"?

Коли необхідна гнучкіший контроль над процесом створення сторінки або незалежність від внутрішніх змін – наприклад, зміна назви сервера не сильно порушить процес побудови відповіді;

9. Яке призначення шаблону «Команда»?

Інкапсулювати запит у вигляді об'єкта, що дозволяє параметризувати клієнтів різними запитами, ставити запити в чергу або протоколювати їх, а також підтримувати скасування операцій.

10. Нарисуйте структуру шаблону «Команда».



11. Які класи входять в шаблон «Команда», та яка між ними взаємодія?

Command (Команда) – абстрактний інтерфейс, який оголошує метод Execute().

ConcreteCommand (Конкретна команда) – Реалізує інтерфейс Command. Він визначає зв'язок між об'єктом-отримувачем (Receiver) і дією. Він зберігає параметри для виклику методу отримувача.

Invoker (Ініціатор) – Ініціює виконання команди. Він зберігає посилання на об'єкт команди і може викликати його метод Execute().

Receiver (Отримувач) – знає, як виконати конкретну операцію. Саме цей клас містить реальну бізнес-логіку.

Client (Клієнт) – створює конкретну команду та встановлює її отримувача.

Client створює об'єкт ConcreteCommand і пов'язує його з об'єктом Receiver. Потім передає створений об'єкт команди Invoker-у. У певний момент (наприклад, після натискання кнопки) Invoker викликає метод Execute() команди. ConcreteCommand виконує необхідні дії, викликаючи один або кілька методів об'єкта Receiver.

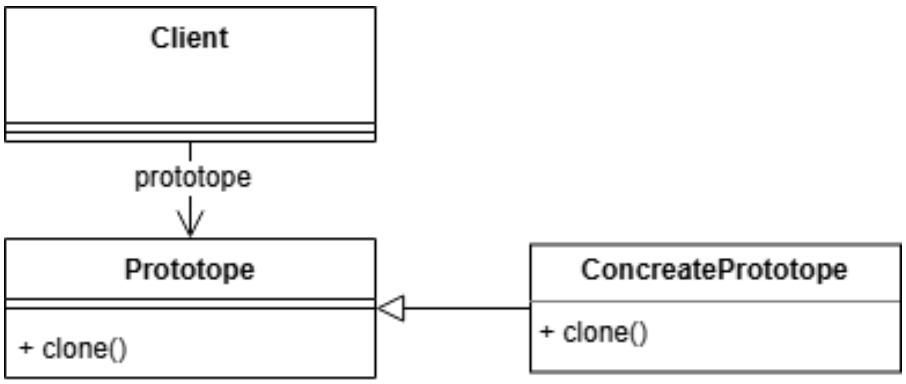
12. Розкажіть як працює шаблон «Команда».

Шаблон «Команда» перетворює запит на об'єкт. Це дозволяє передавати команди як об'єкти, ставити їх у чергу, вести їх історію та реалізовувати скасування операцій.

13. Яке призначення шаблону «Прототип»?

Створювати нові об'єкти шляхом копіювання існуючого об'єкта-прототипу, не роблячи код залежним від їхніх конкретних класів.

14. Нарисуйте структуру шаблону «Прототип».



15. Які класи входять в шаблон «Прототип», та яка між ними взаємодія?

Prototype (Прототип) – оголошує інтерфейс для клонування самого себе.

ConcretePrototype (Конкретний прототип) – реалізує операцію клонування. Він повертає копію самого себе.

Client (Клієнт) – створює новий об'єкт, викликаючи метод Clone() у прототипу.

Client отримує посилання на об'єкт-прототип. Потім викликає метод Clone() цього прототипу. ConcretePrototype створює точну копію себе і повертає її клієнту.

16. Які можна привести приклади використання шаблону «Ланцюжок відповідальності»?

Подія в графічних інтерфейсах (наприклад, клік мишею) передається по ланцюжку: віджету (кнопці), його контейнеру (панелі), головному вікну. Якщо один елемент не може обробити подію, він передає її батьківському елементу.

Логер Системи логування може передавати повідомлення різних рівнів (DEBUG, INFO, WARN, ERROR) по ланцюжку. Наприклад, ConsoleLogger виводить всі повідомлення в консоль, а FileLogger передає повідомлення рівня ERROR і вище для запису в файл. Якщо один логер не обробляє певний рівень, він передає повідомлення наступному в ланцюжку.