Питання до іспиту АППЗ

1. Визначити поняття системної розробки програмного забезпечення.

2. Визначити поняття детального проектування.

3. Визначити поняття вбудованого програмного забезпечення.

4. Обґрунтувати використанню метрик при оцінці якості архітектури програмного забезпечення.

6. Дати оцінку ролі і місцю детального проектування у життєвому циклі програмного забезпечення

7. Дати оцінку цілям вибору архітектури програмного забезпечення.

8. Проаналізувати співвідношення варіантів використання архітектури та детального проектування. (Лекція 6, Співвідношення варіантів використання, архітектури і детального проектування)

Взаємозв'язок між варіантами використання, архітектурою і детальним проектуванням може бути представлений по аналогії з проектуванням мосту. Варіанти використання будуть частиною вимог, що пред'являються до конструкції моста. Відштовхуючись від вимог, розробники вибирають архітектуру (наприклад, підвісний міст). Після цього вони розробляють детальный проект для реалізації необхідних варіантів використання з урахуванням вибраної архітектури (рис. 6.2).

У проектуванні програмного забезпечення кожен наступний рівень накопичує додаткові класи. На кроці 1 варіанти використання фіксуются як частина вимог. На кроці 2 вони разом з іншими джерелами використовуються для визначення класів предметної області. На кроці 3, ми розробляємо програмну архітектуру. Останній крок полягає в перевірці того факту, що архітектура і детальний проект задовольняють необхідним варіантам використання. Стосовно нашого прикладу з проектуванням мосту ми перевіряємо те, чи автомобілі дійсно зможуть використовувати спроектований міст при поїздці з Пагорба Сміта в Долину Джонса, як це було заявлено у варіантах використання. У разі програмного проекту ми перевіряємо те, чи класи і методи, специфіковані при детальному проектуванні, здатні реалізувати необхідні варіанти використання



9. Визначити поняття декомпозиції програмного забезпечення.

10. Описати типову схему детального проектування.

13. Визначити поняття каркасу програмного забезпечення.

14. Дати оцінку використанню повторно використовуваним компонентам у детальному проектуванні.

16. Обґрунтувати використання детальних діаграм послідовності. (Лекція 7, діаграми послідовності)

Діаграми послідовності вважають найпопулярнішим засобом представлення детальних вимог на етапі аналізу. Мета перетворення – підвищити рівень точності, формалізувати вимоги, привести їх до виду, необхідною для розв’язання задач проектування. (<http://eprints.cdu.edu.ua/1465/1/dddd2018.pdf> , ст. 120)

**Лекція.** Частіше всього, діаграми послідовності використовуються як інструмент для проектування, вони є одним зі способів удосконалення моделей для детального проектування. Необхідно виконати наступні дії для здійснення детального проектування:

1. Почніть з діаграм послідовності, створених для детальних вимог або архітектури (якщо такі є), що відносяться до варіантів використання.
2. Якщо це потрібно, представте додаткові варіанти використання, для того, щоб описати, як спроектовані частини зазвичай взаємодіють з іншим додатками.
3. Забезпечте повну деталізацію діаграм послідовності :

* переконайтеся, що відповідні об'єкти і їх класи специфіковані;
* виберіть конкретні імена функцій замість природної мови для представлення операцій.

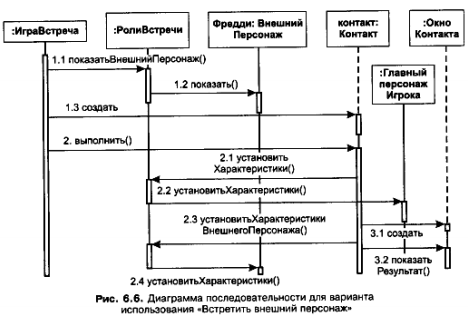
Нагадаємо, що варіанти використання можуть застосовуватися для представлення вимог, і тому ми застосовуємо їх для визначення ключових наслідуваних класів додатка. Тепер візьмемо діаграми послідовності, що побудовані на варіантах використання, і надамо відповідні класи з необхідними для виконання послідовностей методами.

Діаграма послідовності для варіанту використання "Зустріти зовнішній персонаж" гри Зустріч, взята з прикладу у кінці теми, представлена на рис. 6.6. Замість словесних описів функціональності, яку ми використовували для вимог, ми визначили власні функції. Інше удосконалення полягає в представленні класу РоліЗустрічі зразка проектування Facade, через який повинні проходити усі зовнішні посилання персонажі гри. Нижче наводяться причини вибору таких функцій.

1. Класу ЗовнішнійПерсонаж потрібна функція, що забезпечує відображення на екрані. Ми реалізуємо її за допомогою методу показати(). (Оскільки ця можливість потрібна для усіх персонажів, ми можемо гарантувати виконання даної вимоги, додавши метод показати() в базовий клас ГраЗустріч.) На діаграмі послідовності показано, що класом ГраЗустріч створюється зовнішній персонаж (крок 1.2), потім створюється об'єкт Контакт і викликається метод показати().
2. Цей крок у варіанті використання відображає необхідність наявності методу виконати() в класі Контакт. На цьому кроці потрібно, щоб Фреді і головний ПерсонажГравця могли міняти значення своїх характеристик. Оскільки ця можливість є загальною для усіх персонажів Зустрічі, ми закладемо її в базовий клас ПерсонажЗустрічі в методі встановитиХарактеристики().
3. На цьому кроці необхідно показати результат контакту. Одним з можливих рішень є наступне:

1) спочатку Контакт створює об'єкт ВікноКонтакту;

2) потім ми виводимо вікно контакту, викликавши метод показатиРезулътат().



Оскільки всі методи, необхідні для реалізації цього варіанту використання, тепер відомі, ми можемо представити їх на об'єктній моделі (рис. 6.7). Продовжуючи цей процес, ми отримаємо детальну модель класів і модель варіантів використання у вигляді діаграм послідовності. Необхідно також отримати детальну модель переходів станів (якщо вона застосовна). Діаграма потоків даних відноситься до другої моделі і обговорюється далі.



20. Обґрунтувати специфікацію класів та функцій у детальному проектуванні.

21. Навести класифікацію зразків проектування.

22. Визначити поняття інваріантів класів.

23. Обґрунтувати використання компонентів при розробці архітектури програмного забезпечення.

24. Проаналізувати інваріанти передумови та післяумови.

25. Дати оцінку використання моделей при розробці архітектури програмного забезпечення.

26. Визначити прийоми специфікації алгоритмів.

27. Дати оцінку перевагам та недолікам використання програмної архітектури заснованої на потоках даних.

29. Дати оцінку перевагам та недолікам використання програмної архітектури незалежних компонентів.

31. Дати оцінку перевагам та недолікам використання програмної архітектури віртуальних машин.

32. Порівняти переваги та недоліки використання блок-схем та псевдокоду при детальному проектуванні.

33. Дати оцінку перевагам та недолікам використання репозиторної архітектури.

36. Проаналізувати використання креаційних зразків у детальному проектуванні.

38. Проаналізувати використання структурних зразків у детальному проектуванні.

39. Описати інструментальні засоби для розробки програмної архітектури.

40. Проаналізувати використання поведінкових зразків у детальному проектуванні.

41. Описати етапи контролю якості при виборі архітектури.

42. Визначити поняття зразків проектування заснованих на поведінці додатків.

43. Дати оцінку перевірці архітектури за допомогою варіантів використання. (Лекція 5, Перевірка архітектури за допомогою варіантів використання)

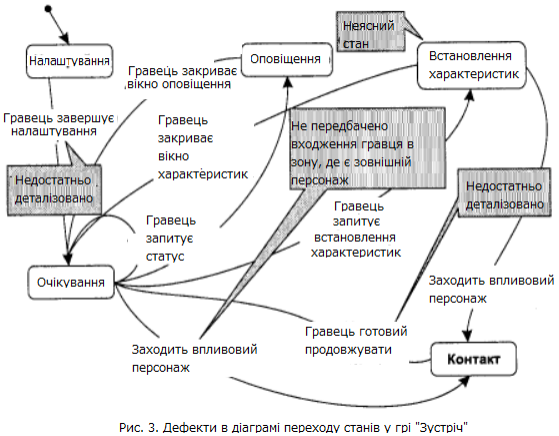
Варіанти використання отримують із вимог замовника. Тому варіанти використання не можуть враховувати архітектуру додатку по простій причині – вона ще не визначена. Після вибору архітектури корисно повернутися до розгляду варіантів використання і перевірити, чи адекватно їх підтримує архітектура. Оскільки впродовж усього проекту ми зберігаємо класи предметної області, класи, на які ми посилалися у варіантах використання, мають бути представлені серед використовуваних. Зазвичай діаграми послідовності для варіантів використання на цьому етапі включають додаткові класи архітектури.

45. Описати етапи інспектування вибору архітектури. (Лекція 5, Інспектування вибору архітектури)

Архітектура перевіряється на відповідність вимогам. Нагадаємо, що виграш в часі і засобах максимальний, якщо дефект виявлений і усунений на початковій стадії проекту, наприклад на рівні вибору архітектури. Метрики, розглянуті раніше, складають один із базисів інспекції архітектури.

Інспекція пакетів каркаса архітектури для гри Зустріч може принести до виводу, що ще немає ніяких вимог для появи пакету Артефакти і що цей пакет і є власне дефектом.

В якості додаткового прикладу візьмемо діаграму переходів станів, що зображена на рис. 3. Уважний розгляд діаграми переходів станів може виявити дефекти, показані на малюнку. Ці дефекти можна усунути шляхом уточнення імен і (або) визначення подій, на які виконуються посилання.



47. Перерахуйте основні стандарти для документування проекту програмного продукту. (Лекція 4, Єдина система програмної документації)

**!!! Можливо, тут він хоче почути про стандарт IEEE/ANSI для опису проекту (теж 4 лекція), або Єдина система програмної документації, мені здається що друге.**

Основу вітчизняної нормативної бази в області документування ПЗ становить комплекс стандартів Єдиної системи програмної документації. (в лекції є більше тексту, якщо потрібно).

Перелік документів ЕСПД (ЄСПД?)

1. ГОСТ 19.001-77 ЕСПД. Загальні положення.

2. ГОСТ 19.101-77 ЕСПД. Види програм і програмних документів.

3. ГОСТ 19.102-77 ЕСПД. Стадії розробки.

4. ГОСТ 19.103-77 ЕСПД. Позначення програм і програмних документів.

5. ГОСТ 19.104-78 ЕСПД. Основні написи.

6. ГОСТ 19.105-78 ЕСПД. Загальні вимоги до програмних документів.

7. ГОСТ 19.106-78 ЕСПД. Вимоги до програмних документів, виконаним друкованим способом.

8. ГОСТ 19.201-78 ЕСПД. Технічне завдання. Вимоги до змісту та оформлення.

9. ГОСТ 19.202-78 ЕСПД. Специфікація. Вимоги до змісту та оформлення.

10. ГОСТ 19.301-79 ЕСПД. Порядок і методика випробувань.

11. ГОСТ 19.401-78 ЕСПД. Текст програми. Вимоги до змісту та оформлення.

12. ГОСТ 19.402-78 ЕСПД. Опис програми.

13. ГОСТ 19.404-79 ЕСПД. Пояснювальна записка. Вимоги до змісту та оформлення.

14. ГОСТ 19.501-78 ЕСПД. Формуляр. Вимоги до змісту та оформлення.

15. ГОСТ 19.502-78 ЕСПД. Опис застосування. Вимоги до змісту та оформлення.

16. ГОСТ 19.503-79 ЕСПД. Керівництво системного програміста. Вимоги до змісту та оформлення.

17. ГОСТ 19.504-79 ЕСПД. Керівництво програміста.

18. ГОСТ 19.505-79 ЕСПД. Керівництво оператора.

19. ГОСТ 19.506-79 ЕСПД. Опис мови.

20. ГОСТ 19.508-79 ЕСПД. Керівництво з технічного обслуговування. Вимоги до змісту та оформлення.

21. ГОСТ 19.604-78 ЕСПД. Правила внесення змін у програмні документи, що їх друкарським способом.

22. ГОСТ 19.701-90 ЕСПД. Схеми алгоритмів, програм, даних і систем. Умовні позначення і правила виконання.

23. ГОСТ 19.781-90. Забезпечення систем обробки інформації програмне.

48. Проаналізувати співвідношення варіантів використання архітектури та детального проектування. (Лекція 6, Співвідношення варіантів використання, архітектури і детального проектування)

Взаємозв'язок між варіантами використання, архітектурою і детальним проектуванням може бути представлений по аналогії з проектуванням мосту. Варіанти використання будуть частиною вимог, що пред'являються до конструкції моста. Відштовхуючись від вимог, розробники вибирають архітектуру (наприклад, підвісний міст). Після цього вони розробляють детальный проект для реалізації необхідних варіантів використання з урахуванням вибраної архітектури (рис. 6.2).

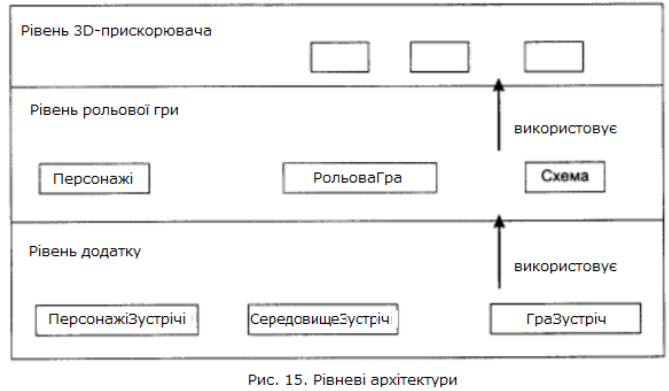
У проектуванні програмного забезпечення кожен наступний рівень накопичує додаткові класи. На кроці 1 варіанти використання фіксуются як частина вимог. На кроці 2 вони разом з іншими джерелами використовуються для визначення класів предметної області. На кроці 3, ми розробляємо програмну архітектуру (див. Теми 1-7). Останній крок полягає в перевірці того факту, що архітектура і детальний проект задовільняють необхідним варіантам використання. Стосовно нашого прикладу з проектуванням мосту ми перевіряємо те, чи автомобілі дійсно зможуть використовувати спроектований міст при поїздці з Пагорба Сміта в Долину Джонса, як це було заявлено у варіантах використання. У разі програмного проекту ми перевіряємо те, чи класи і методи, специфіковані при детальному проектуванні, здатні реалізувати необхідні варіанти використання



49. Навести переваги використання рівневих архітектур. (Лекція 3, Рівнева архітектура, в лекції є ще більше діаграм та тексту)

Рівень архітектури – це логічно пов'язана колекція артефактів програмного забезпечення, зазвичай – пакетів класів. У загальному вигляді рівень використовує не більше одного рівня і в той же час використовується не більше ніж одним рівнем. Побудова додатків послідовна рівень за рівнем сильно спрощує процес. Деякі рівні, наприклад каркаси, можуть використовуватися в декількох додатках.

Приклад рівневої архітектури – використання прискорювача тривимірної графіки як рівня, доступного з рівня ролевої гри Зустріч, – показаний на рис. 15.



Клієнт-серверна архітектура є загальною рівневою формою. У цій формі рівень клієнта зв'язується з серверним рівнем для отримання потрібного сервісу. Клієнт зазвичай знаходиться резидентно на комп'ютері користувача, а сервер – на більшому централізованому комп'ютері. Сервер нерідко посилається на базу даних.

Класична клієнт-серверна архітектура зазвичай страждає від реалізації. Річ у тому, що частенько клієнти і сервери закодовані на рівні технічних засобів у високій мірі залежності один від одного. Ця проблема може бути розв'язана за допомогою трирівневої архітектури, в якій середній рівень призначений для розподілу клієнта і сервера. Середній рівень можна використовувати для збільшення гнучкості архітектури декількома способами. Наприклад, якщо декілька серверів одночасно можуть обслужити запит клієнта, то середній рівень може динамічно визначити відповідний сервер. Средній рівень зазвичай реалізується у вигляді єднального програмного забезпечення (middleware).

Рівнева архітектура має безліч переваг при повторному використанні. Бібліотека класів Java є дуже ефективною рівневою системою (пакет applet - рівень - заснований на пакеті awt, який у свою чергу заснований на пакеті lang. і т. д.)

50. Обґрунтувати використання детальних діаграм послідовності.

51. Визначити поняття створення архітектури програмного забезпечення.

52. Обґрунтувати використання блок-схем у детальному проектуванні.

53. Описати процедуру вибору архітектури.

54. Дати оцінку основним прийомам детального проектування.

55. Визначити поняття зразка проектування програмного забезпечення.

56. Обґрунтувати використання псевдокоду у детальному проектуванні.

57. Навести класифікацію архітектур програмного забезпечення.

58. Проаналізувати використання інтерфейсів у детальному проектуванні.

59. Дати оцінку використання каркасів у проектуванні програмного забезпечення

60. Обґрунтувати використання детальних діаграм потоків даних.

61. Подати за допомогою діаграми класів структуру патерна Спостерігач (Observer), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

62. Подати за допомогою діаграми класів структуру патерна Команда (Command), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

63. Подати за допомогою діаграми класів структуру патерна Ітератор (Iterator), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

64. Подати за допомогою діаграми класів структуру патерна Пристосуванець (Flyweight), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

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

В якості стандартного застосування даного патерну можна привести наступний приклад. Текст складається з окремих символів. Кожен символ може зустрічатися на одній сторінці тексту багато разів. Однак в комп'ютерній програмі було б занадто накладно виділяти пам'ять для кожного окремого символу в тексті. Набагато простіше було б визначити повний набір символів, наприклад, у вигляді таблиці з 128 знаків (алфавітно-цифрові символи в різних регістрах, розділові знаки і т.д.). А в тексті застосувати цей набір загальних символів, що розділяються, замість сотень і тисяч об'єктів, які могли б використовуватися в тексті. І як наслідок подібного підходу буде зменшення кількості використовуваних об'єктів і зменшення використовуваної пам'яті. Патерн пристосуванець слід застосовувати при дотриманні всіх наступних умов:

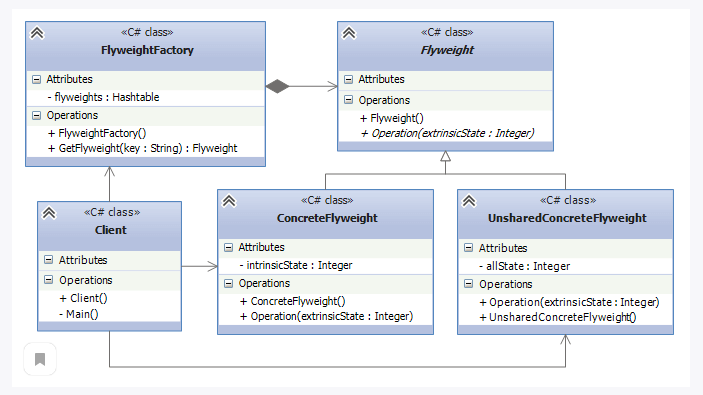
- Коли додаток використовує велику кількість одноманітних об'єктів, через що відбувається виділення великої кількості пам'яті

- Коли частина стану об'єкта, який є змінним, можна винести. Винесення зовнішнього стану дозволяє замінити безліч об'єктів невеликою групою спільних об'єктів.

Ключовим моментом тут є поділ стану на внутрішньо і зовнішнє. Внутрішній стан не залежить від контексту. У прикладі з символами внутрішній стан описується кодом символу з таблиці кодування. Так як внутрішній стан не залежить від контексту, то воно може бути поділюваним і тому виноситься в розділяються об'єкти.

Зовнішній стан залежить від контексту, є мінливим. У застосуванні до символів зовнішній стан може представляти положення символу на сторінці. Тобто код символу може бути використаний багатьма символами, тоді як положення на сторінці буде для кожного символу індивідуально.

При створенні пристосуванця зовнішній стан виноситься. У пристосуванці залишається тільки внутрішній стан. Тобто в прикладі з символами пристосуванець буде зберігати код символу. Відносини в даному патерні можна описати наступною схемою:



Flyweight: визначає інтерфейс, через який пристосуванці-розділювані об'єкти можуть отримувати зовнішній стан або впливати на нього

ConcreteFlyweight: конкретний клас поділяється пристосуванця. Реалізує інтерфейс, оголошений в типі Flyweight, і при необхідності додає внутрішній стан. Причому будь-яке зберігається їм стан має бути внутрішнім, не залежним від контексту.

UnsharedConcreteFlyweight: ще одна конкретна реалізація інтерфейсу, визначеного в типі Flyweight, тільки тепер об'єкти цього класу є нероздільними

FlyweightFactory: фабрика пристосуванців - створює об'єкти поділюваних пристосуванців. Так як пристосуванці поділяються, то клієнт не повинен створювати їх безпосередньо. Всі створені об'єкти зберігаються в Пулі. У прикладі вище для визначення пулу використовується Об'єкт Hashtable, але це не обов'язково. Можна застосовувати і інші класи колекцій. Однак в залежності від складності структури, що зберігає розділяються об'єкти, особливо якщо у нас велика кількість пристосуванців, то може збільшуватися час на пошук потрібного пристосуванця - напевно це один з небагатьох недоліків даного патерну. Якщо запитаного пристосуванця не виявилося в Пулі, то фабрика створює його.

Client: використовує об'єкти пристосуванців. Може зберігати зовнішній стан і передавати його в якості аргументів в методи пристосуванців.

Слід використовувати, якщо не вистачає оперативної пам’яті для підтримки всіх потрібних об’єктів.

Ефективність патерну Легковаговик багато в чому залежить від того, як і де він використовується. Застосовуйте цей патерн у випадках, коли: у програмі використовується велика кількість об’єктів;

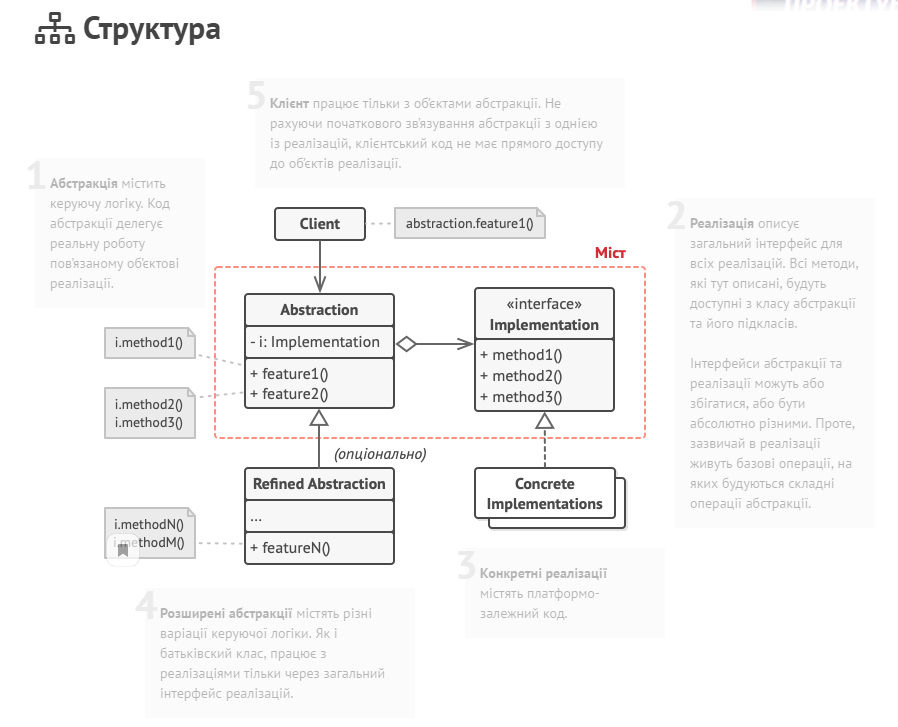
1. через це високі витрати оперативної пам’яті;
2. більшу частину стану об’єктів можна винести за межі їхніх класів;
3. великі групи об’єктів можна замінити невеликою кількістю об’єктів, що розділяються, оскільки зовнішній стан винесено.

65. Подати за допомогою діаграми класів структуру патерна Міст (Bridge), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

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

Отже, абстракція (або інтерфейс) — це уявний рівень керування чим-небудь, що не виконує роботу самостійно, а делегує її рівню реалізації (який зветься платформою).

Якщо говорити про реальні програми, то абстракцією може виступати графічний інтерфейс програми (GUI), а реалізацією — низькорівневий код операційної системи (API), до якого графічний інтерфейс звертається, реагуючи на дії користувача.



Застосування

1. Якщо ви хочете розділити монолітний клас, який містить кілька різних реалізацій якої-небудь функціональності (наприклад, якщо клас може працювати з різними системами баз даних).

Чим більший клас, тим важче розібратись у його коді, і тим більше це розтягує час розробки. Крім того, зміни, що вносяться в одну з реалізацій, призводять до редагування всього класу, що може викликати появу несподіваних помилок у коді.

Міст дозволяє розділити монолітний клас на кілька окремих ієрархій. Після цього ви можете змінювати код в одній гілці класів незалежно від іншої. Це спрощує роботу над кодом і зменшує ймовірність внесення помилок.

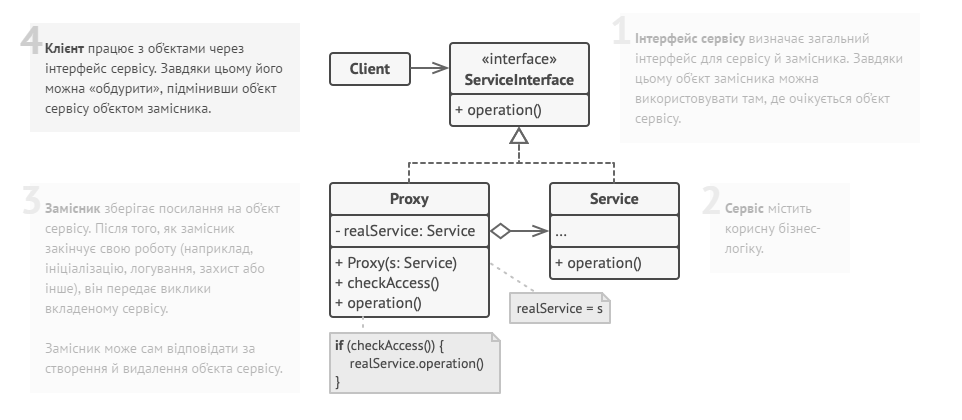
1. Якщо клас потрібно розширювати в двох незалежних площинах.

Міст пропонує виділити одну з таких площин в окрему ієрархію класів, зберігаючи посилання на один з її об’єктів у початковому класі.

1. Якщо ви хочете мати можливість змінювати реалізацію під час виконання програми. Міст дозволяє замінювати реалізацію навіть під час виконання програми, оскільки конкретна реалізація не «зашита» в клас абстракції.

66. Подати за допомогою діаграми класів структуру патерна Замісник (Proxy), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

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



Застосування

1. Лінива ініціалізація (віртуальний проксі). Коли у вас є важкий об’єкт, який завантажує дані з файлової системи або бази даних.

Замість того, щоб завантажувати дані відразу після старту програми, можна заощадити ресурси й створити об’єкт тоді, коли він дійсно знадобиться.

1. Захист доступу (захищаючий проксі). Коли в програмі є різні типи користувачів, і вам хочеться захистити об’єкт від неавторизованого доступу. Наприклад, якщо ваші об’єкти — це важлива частина операційної системи, а користувачі — сторонні програми (корисні чи шкідливі).

Проксі може перевіряти доступ під час кожного виклику та передавати виконання службовому об’єкту, якщо доступ дозволено.

1. Локальний запуск сервісу (віддалений проксі). Коли справжній сервісний об’єкт знаходиться на віддаленому сервері.

У цьому випадку замісник транслює запити клієнта у виклики через мережу по протоколу, який є зрозумілим віддаленому сервісу.

1. Логування запитів (логуючий проксі). Коли потрібно зберігати історію звернень до сервісного об’єкта.

Замісник може зберігати історію звернення клієнта до сервісного об’єкта.

1. Кешування об’єктів («розумне» посилання). Коли потрібно кешувати результати запитів клієнтів і керувати їхнім життєвим циклом.

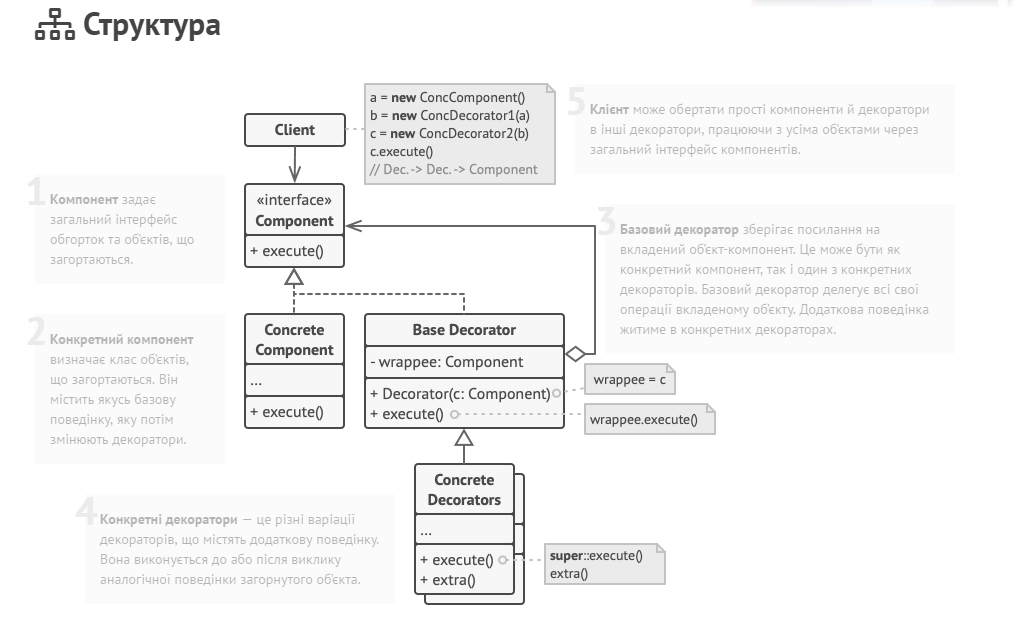
Замісник може підраховувати кількість посилань на сервісний об’єкт, які були віддані клієнту та залишаються активними. Коли всі посилання звільняться, можна буде звільнити і сам сервісний об’єкт (наприклад, закрити підключення до бази даних).

1. Крім того, Замісник може відстежувати, чи клієнт не змінював сервісний об’єкт. Це дозволить повторно використовувати об’єкти й суттєво заощаджувати ресурси, особливо якщо мова йде про великі «ненажерливі» сервіси.

67. Подати за допомогою діаграми класів структуру патерна Декоратор (Decorator), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

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

Для визначення нового функціоналу в класах нерідко використовується спадкування. Декоратори ж надає спадкуванню більш гнучку альтернативу, оскільки дозволяють динамічно в процесі виконання визначати нові можливості у об'єктів.



Застосування

1. Якщо вам потрібно додавати об’єктам нові обов’язки «на льоту», непомітно для коду, який їх використовує.

Об’єкти вкладаються в обгортки, які мають додаткові поведінки. Обгортки і самі об’єкти мають однаковий інтерфейс, тому клієнтам не важливо, з чим працювати — зі звичайним об’єктом чи з загорнутим.

1. Якщо не можна розширити обов’язки об’єкта за допомогою спадкування.

У багатьох мовах програмування є ключове слово final, яке може заблокувати спадкування класу. Розширити такі класи можна тільки за допомогою Декоратора.

68. Подати за допомогою діаграми класів структуру патерна Адаптер (Adapter), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

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

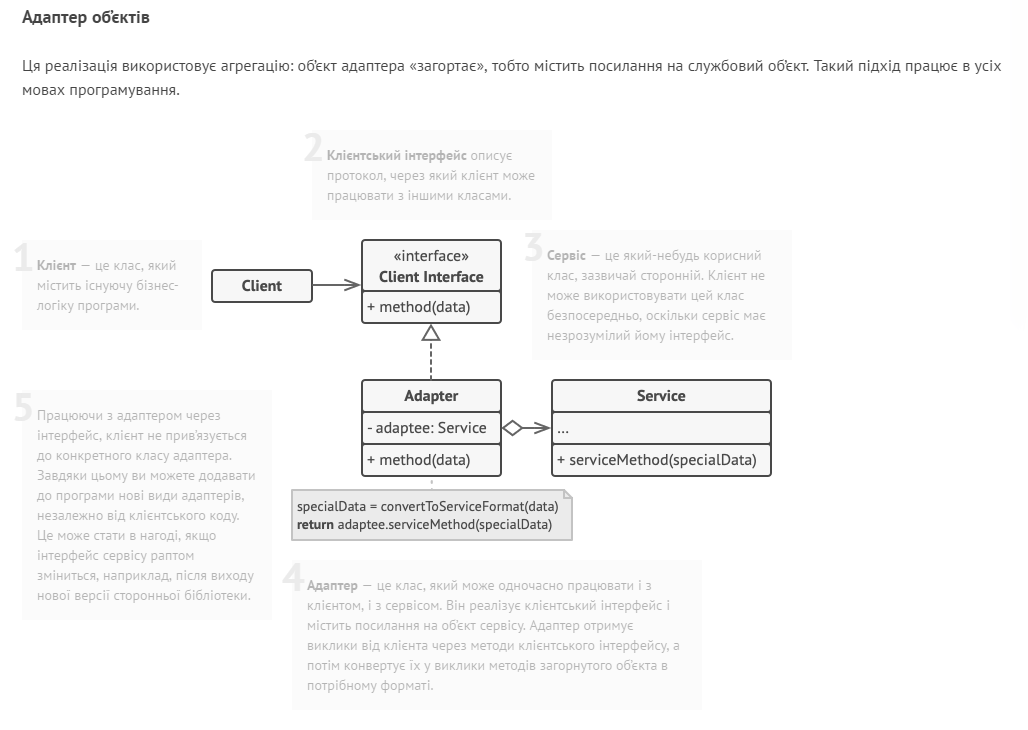
Адаптер загортає один з об’єктів так, що інший об’єкт навіть не підозрює про існування першого. Наприклад, об’єкт, що працює в метричній системі вимірювання, можна «обгорнути» адаптером, який буде конвертувати дані у фути.

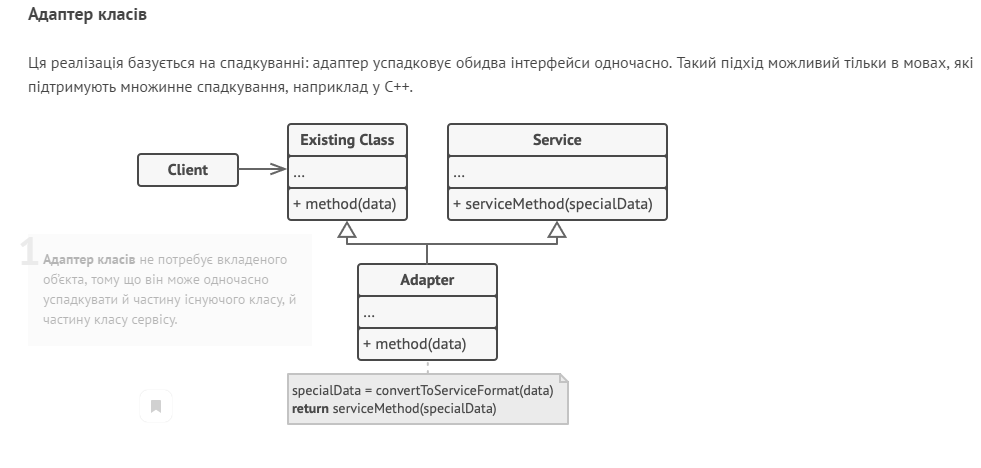
Адаптери можуть не тільки конвертувати дані з одного формату в іншій, але й допомагати об’єктам із різними інтерфейсами працювати разом. Це виглядає так:

Адаптер має інтерфейс, сумісний з одним із об’єктів.

Тому цей об’єкт може вільно викликати методи адаптера.

Адаптер отримує ці виклики та перенаправляє їх іншому об’єкту, але вже в тому форматі та послідовності, які є зрозумілими для цього об’єкта.





Застосування

1. Якщо ви хочете використати сторонній клас, але його інтерфейс не відповідає решті кодів програми.

Адаптер дозволяє створити об’єкт-прокладку, який перетворюватиме виклики програми у формат, зрозумілий сторонньому класу.

1. Якщо вам потрібно використати декілька існуючих підкласів, але в них не вистачає якої-небудь спільної функціональності, а розширити суперклас ви не можете.

Ви могли б створити ще один рівень підкласів та додати до них забраклу функціональність. Але при цьому доведеться дублювати один і той самий код в обох гілках підкласів.

69. Подати за допомогою діаграми класів структуру патерна Інформаційний експерт (Information Expert) , коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

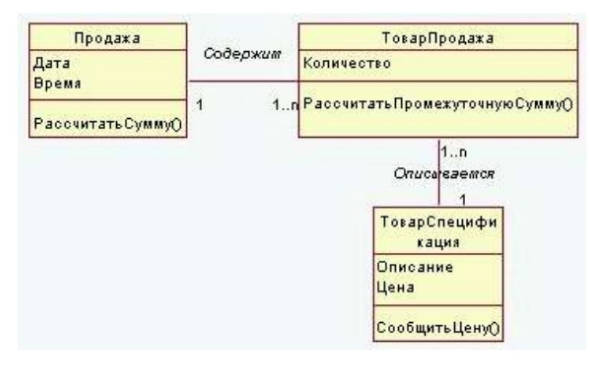
GRASP (General Responsibility Assignment Software Patterns) – загальні зразки розподілу обов'язків GRASP-патерни, що використовуються в об'єктно-орієнтованому проектуванні для вирішення спільних завдань за призначенням обов'язків класам і об'єктам.

Один із них Інформаційний експерт – описує основоположні принципи призначення обов'язків класам і об'єктам. Згідно з описом, інформаційним експертом (об'єктом наділеним деякими обов'язками) є об'єкт, що володіє максимумом інформацією, необхідною для виконання призначених обов'язків. – Інформаційний експерт.

Проблема: в системі повинна акумулюватися, розраховуватися і т.п. необхідна інформація.

Рішення: призначити обов'язок акумуляції інформації, розрахунку і т.п. якомусь класу (інформаційному експерту), що володіє необхідною інформацією.

Рекомендації: інформаційним експертом може бути не один клас, а кілька.



Переваги:

Підтримує інкапсуляцію, тобто об'єкти використовують свої власні дані для виконання поставлених завдань

Поведінка системи забезпечується класами, що містять необхідну інформацію. Підтримується High Cohesion (висока зв’язаність)

Недоліки: При проблемах зі зв'язуванням і зачепленням.

70. Подати за допомогою діаграми класів структуру патерна Низька зв’язність (Low Coupling), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

Низька зв'язаність (Low Coupling) і високе зачеплення (High Cohesion)

Мабуть в будь-якій літературі з об'єктно-орієнтованого проектування зустрічаються ці два поняття. Вважається, що будь-яка спроектована система, повинна задовольняти принципам низької зв'язності і високого зачеплення модулів. Відповідність даним шаблонам дозволяє легко модифікувати і супроводжувати програмний код а також підвищує ступінь його повторного використання.

Розглянемо поняття міри зв'язності модулів і заходи зачеплення модуля. Міра зв'язності модулів визначається кількістю інформації якої має в своєму розпорядженні один модуль про природу іншого. У свою чергу, міра зачеплення модуля визначається ступенем сфокусованості його обов'язків.

Варто відзначити, що існують методології, згідно з якими заходи зв'язності і зачеплення можна оцінити за шкалою від 1 до 10 для конкретного випадку. Однак, в рамках даної статті вони на розглядаються.

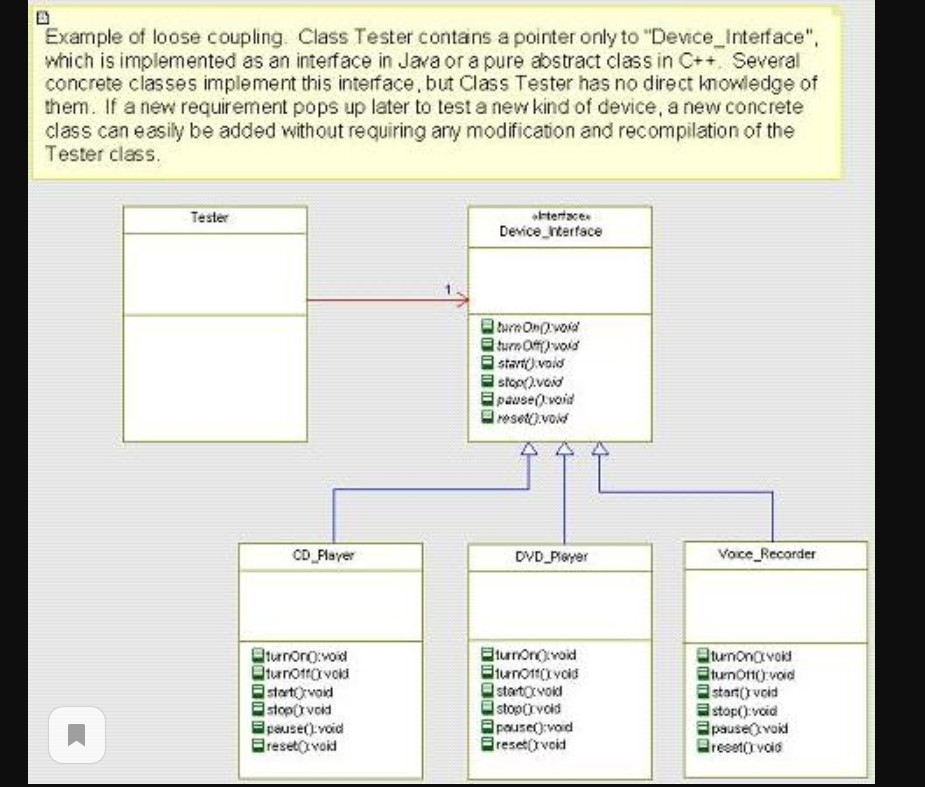
Прикладом гарного дизайну системи може служити набір утиліт GNU Binutils для Linux. В якому, кожна утиліта (якщо її розглядати як модуль) виконує лише мінімальні обов'язки (високе зачеплення) і майже нічого не знає про природу інших утиліт (низька зв'язність), в зв'язку з чим може бути легко замінена на аналог в деякому варіанті використання.

Низька зв'язаність спонукає вас призначити відповідальність так, щоб її розміщення не збільшувало зв'язаність до рівня, який призводить до негативних результатів, які може дати високе зчеплення.

! Низька зв'язаність підтримує конструкцію класів, які є більш незалежними, що знижує вплив змін.

! Низька зв'язаність - це принцип, який слід враховувати при прийнятті всіх проектних рішень;

! Це оцінний принцип, який ви застосовуєте при оцінці всіх проектних рішень.



71. Подати за допомогою діаграми класів структуру патерна Фасад (Facade), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

72. Подати за допомогою діаграми класів структуру патерна Компоновщик (Composite), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

73. Подати за допомогою діаграми класів структуру патерна Інтерпретатор (Interpreter), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

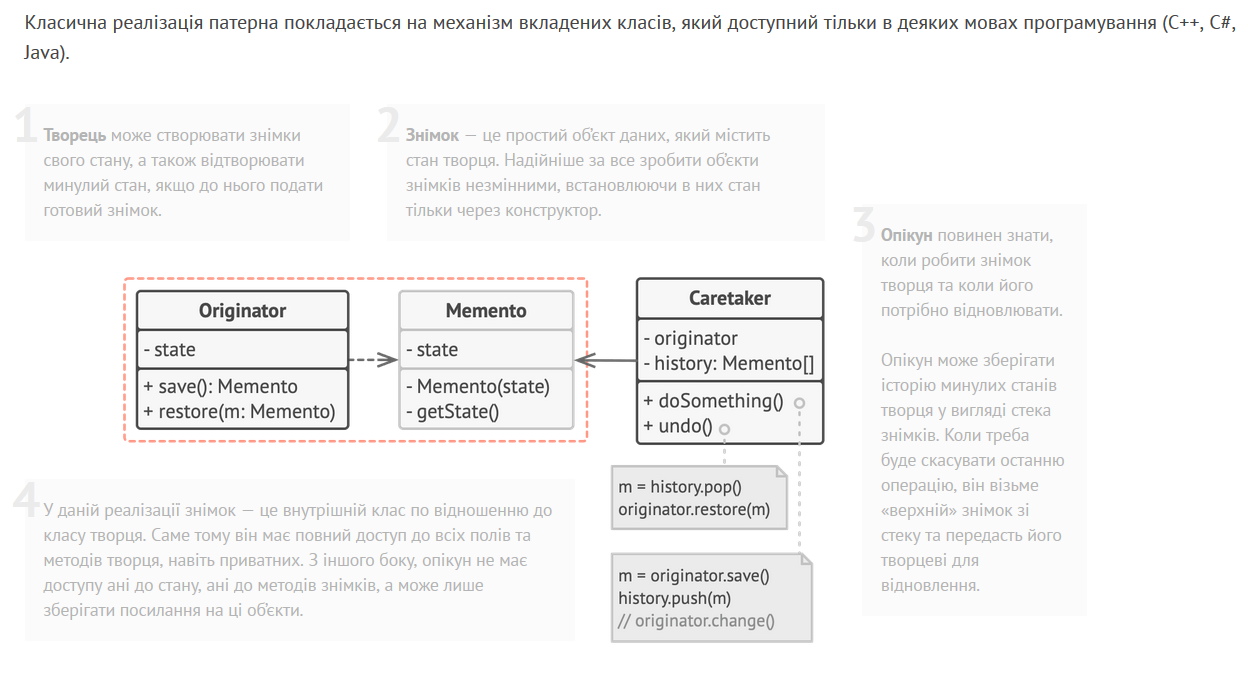
74. Подати за допомогою діаграми класів структуру патерна Відвідувач (Visitor), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

75. Подати за допомогою діаграми класів структуру патерна Посередник (Mediator), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

76. Подати за допомогою діаграми класів структуру патерна Стан (State), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

77. Подати за допомогою діаграми класів структуру патерна Стратегія (Strategy), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

78. Подати за допомогою діаграми класів структуру патерна Зберігач (Memento), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування



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

**Застосування**

* Коли вам потрібно зберігати миттєві знімки стану об’єкта (або його частини) для того, щоб об’єкт можна було відновити в тому самому стані.
* Патерн Знімок дозволяє створювати будь-яку кількість знімків об’єкта і зберігати їх незалежно від об’єкта, з якого роблять знімок. Знімки часто використовують не тільки для реалізації операції скасування, але й для транзакцій, коли стан об’єкта потрібно «відкотити», якщо операція не була вдалою.
* Коли пряме отримання стану об’єкта розкриває приватні деталі його реалізації, порушуючи інкапсуляцію.
* Патерн пропонує виготовити знімок саме вихідному об’єкту, тому що йому доступні всі поля, навіть приватні.

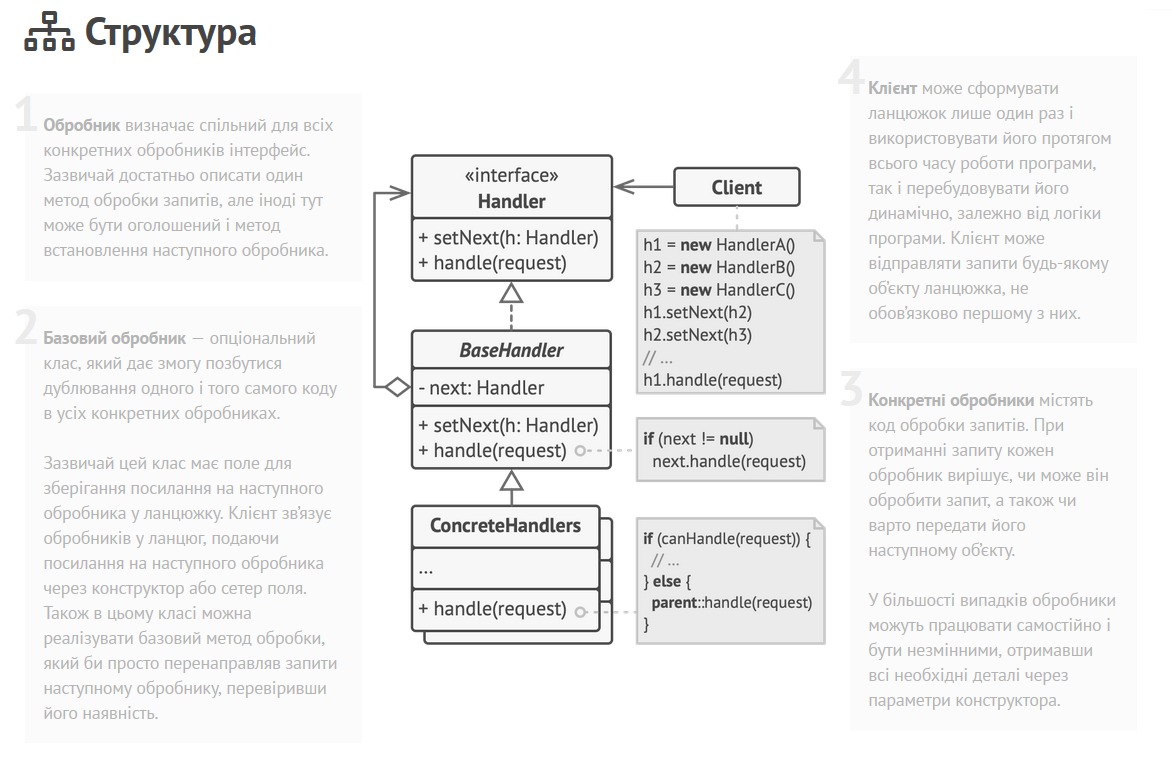
**Переваги:**

* Не порушує інкапсуляцію вихідного об’єкта.
* Спрощує структуру вихідного об’єкта. Йому не потрібно зберігати історію версій свого стану.

**Недоліки:**

* Вимагаєбагатопам’яті, якщоклієнтидужечастостворюютьзнімки.
* Можеспричинитидодатковівитратипам’яті, якщооб’єкти, щозберігаютьісторію, незвільняютьресурси, зайнятізастарілимизнімками.
* Вдеякихмовах (наприклад, PHP, Python, JavaScript) складногарантувати, щоблишевихіднийоб’єктмавдоступдостанузнімка.

79. Подати за допомогою діаграми класів структуру патерна Ланцюжок обов’язків (Chain of Responsibility), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування



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

## Застосування

* Якщо програма має обробляти різноманітні запити багатьма способами, але заздалегідь невідомо, які конкретно запити надходитимуть і які обробники для них знадобляться.
* За допомогою Ланцюжка обов’язків ви можете зв’язати потенційних обробників в один ланцюг і по отриманню запита по черзі питати кожного з них, чи не хоче він обробити даний запит.
* Якщо важливо, щоб обробники виконувалися один за іншим у суворому порядку.
* Ланцюжок обов’язків дозволяє запускати обробників один за одним у тій послідовності, в якій вони стоять в ланцюзі.
* Якщо набір об’єктів, здатних обробити запит, повинен задаватися динамічно.
* У будь-який момент ви можете втрутитися в існуючий ланцюжок і перевизначити зв’язки так, щоби прибрати або додати нову ланку.

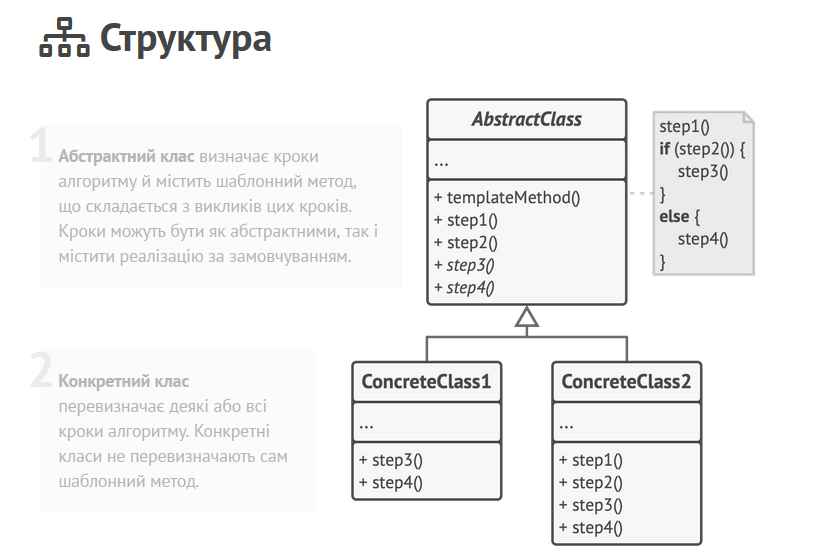
Переваги:

* Зменшує залежність між клієнтом та обробниками.
* Реалізує *принцип єдиного обов’язку*.
* Реалізує *принцип відкритості/закритості*.

Недоліки:

* Запит може залишитися ніким не опрацьованим.

80. Подати за допомогою діаграми класів структуру патерна Шаблонний метод (Template Method), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування



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

## Застосування

* Якщо підкласи повинні розширювати базовий алгоритм, не змінюючи його структури.
* Шаблонний метод дозволяє підкласами розширювати певні кроки алгоритму через спадкування, не змінюючи при цьому структуру алгоритмів, оголошену в базовому класі.
* Якщо у вас є кілька класів, які роблять одне й те саме з незначними відмінностями. Якщо ви редагуєте один клас, тоді доводиться вносити такі ж виправлення до інших класів.
* Патерн шаблонний метод пропонує створити для схожих класів спільний суперклас та оформити в ньому головний алгоритм у вигляді кроків. Кроки, які відрізняються, можна перевизначити у підкласах.
* Це дозволить прибрати дублювання коду в кількох класах, які відрізняються деталями, але мають схожу поведінку.

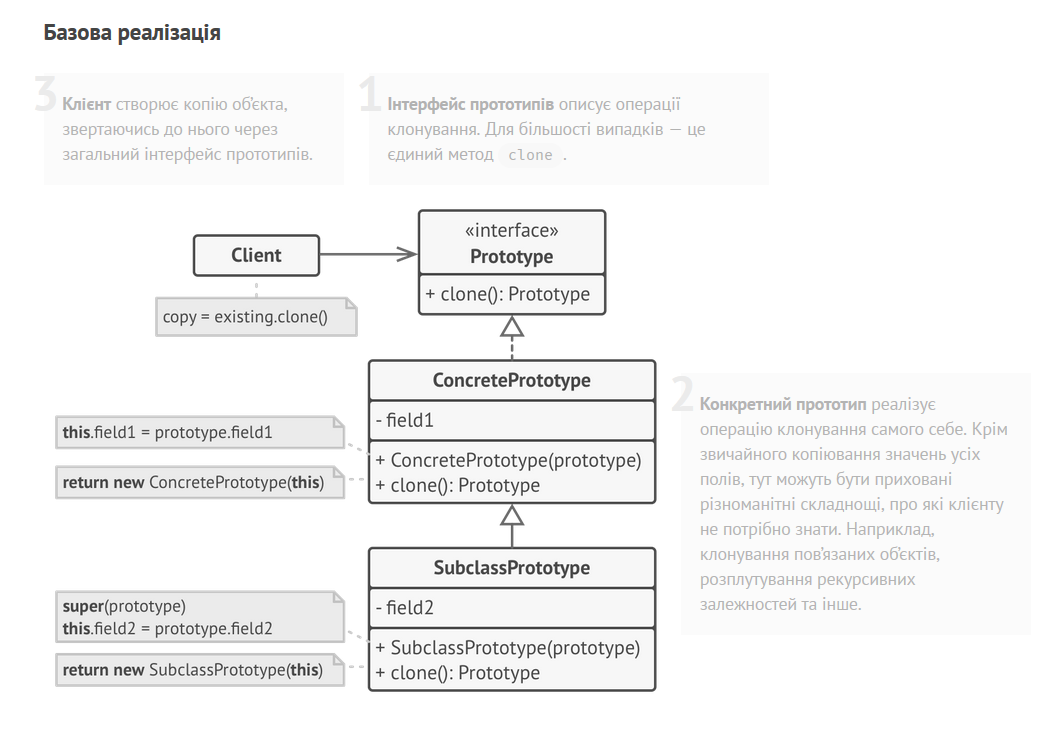
Переваги :

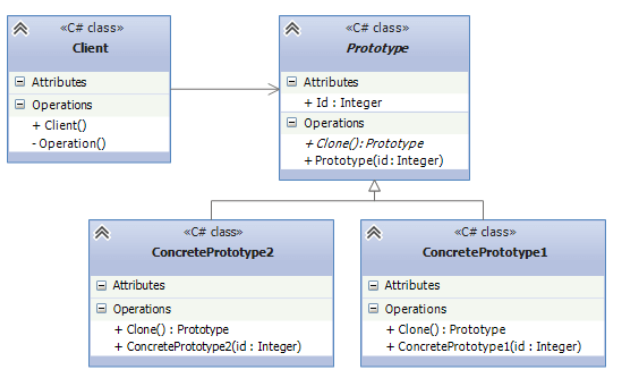
* Полегшує повторне використання коду.

Недоліки:

* Ви жорстко обмежені скелетом існуючого алгоритму.
* Ви можете порушити принцип підстановки Барбари Лісков, змінюючи базову поведінку одного з кроків алгоритму через підклас.
* У міру зростання кількості кроків шаблонний метод стає занадто складно підтримувати.

81. Подати за допомогою діаграми класів структуру патерна Прототип (Prototype), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування





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

## Застосування

* Коли ваш код не повинен залежати від класів об’єктів, призначених для копіювання.
* Таке часто буває, якщо ваш код працює з об’єктами, поданими ззовні через який-небудь загальний інтерфейс. Ви не зможете прив’язатися до їхніх класів, навіть якби захотіли, тому що конкретні класи об’єктів невідомі.
* Патерн Прототип надає клієнту загальний інтерфейс для роботи з усіма прототипами. Клієнту не потрібно залежати від усіх класів об’єктів, призначених для копіювання, а тільки від інтерфейсу клонування.
* Коли ви маєте безліч підкласів, які відрізняються початковими значеннями полів. Хтось міг створити усі ці класи для того, щоб мати легкий спосіб породжувати об’єкти певної конфігурації.
* Патерн Прототип пропонує використовувати набір прототипів замість створення підкласів для опису популярних конфігурацій об’єктів.
* Таким чином, замість породження об’єктів з підкласів ви копіюватимете існуючі об’єкти-прототипи, внутрішній стан яких вже налаштовано. Це дозволить уникнути вибухоподібного зростання кількості класів програми й зменшити її складність.

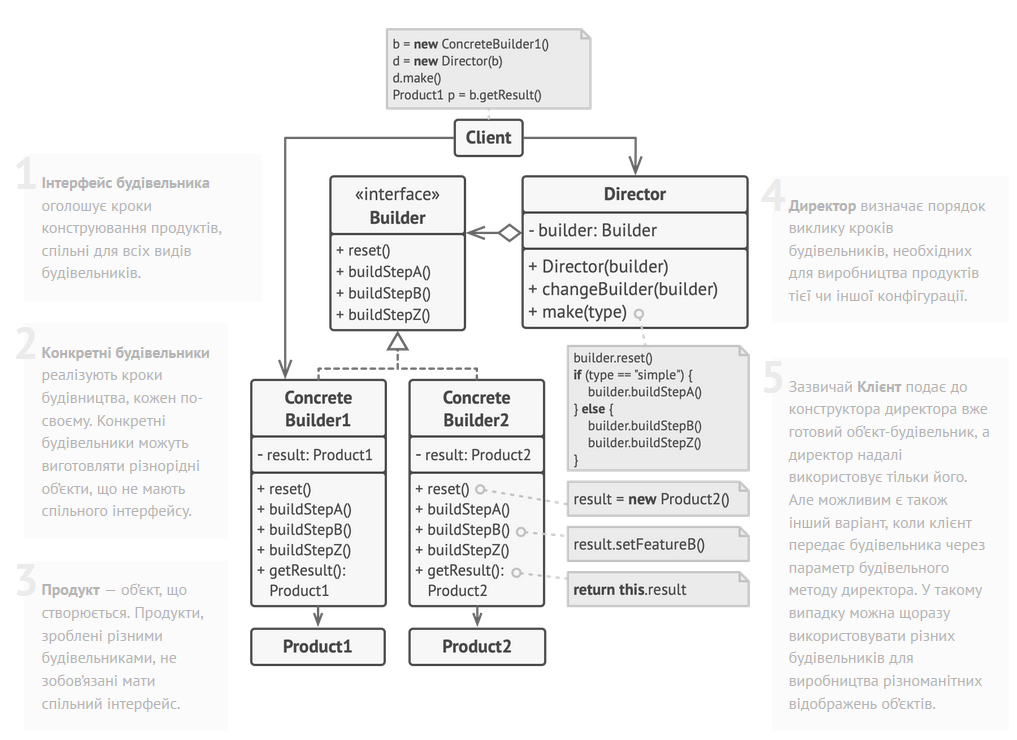
Переваги :

* Дозволяє клонувати об’єкти без прив’язки до їхніх конкретних класів.
* Менша кількість повторювань коду ініціалізації об’єктів.
* Прискорює створення об’єктів.
* Альтернатива створенню підкласів під час конструювання складних об’єктів.

Недоліки:

* Складно клонувати складові об’єкти, що мають посилання на інші об’єкти.

82. Подати за допомогою діаграми класів структуру патерна Будівельник (Builder), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування



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

**Застосування**

* Коли ви хочете позбутися від «телескопічного конструктора».
* Коли ваш код повинен створювати різні уявлення якогось об’єкта. Наприклад, дерев’яні та залізобетонні будинки.
* Будівельник можна застосувати, якщо створення кількох відображень об’єкта складається з однакових етапів, які відрізняються деталями.
* Інтерфейс будівельників визначить всі можливі етапи конструювання. Кожному відображенню відповідатиме власний клас-будівельник. Порядок етапів будівництва визначатиме клас-директор.
* Коли вам потрібно збирати складні об’єкти, наприклад, дерева [Компонувальника](https://refactoring.guru/uk/design-patterns/composite).
* Будівельник конструює об’єкти покроково, а не за один прохід. Більш того, кроки будівництва можна виконувати рекурсивно. А без цього не побудувати деревоподібну структуру на зразок [Компонувальника](https://refactoring.guru/uk/design-patterns/composite).
* Зауважте, що Будівельник не дозволяє стороннім об’єктам отримувати доступ до об’єкта, що конструюється, доки той не буде повністю готовий. Це захищає клієнтський код від отримання незавершених «битих» об’єктів.

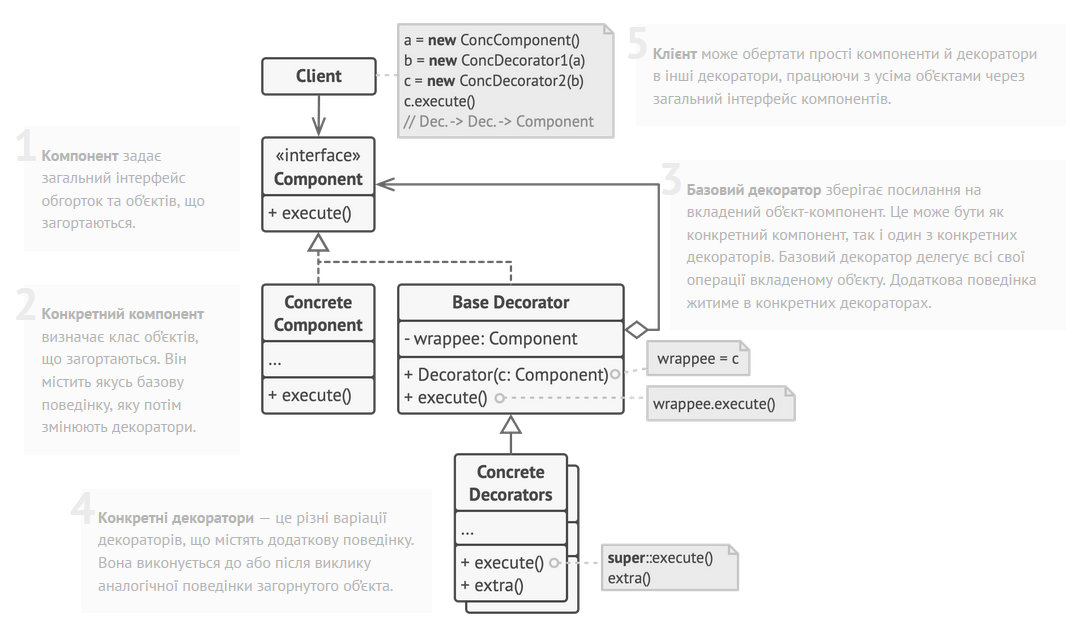
Переваги:

* Дозволяє створювати продукти покроково.
* Дозволяє використовувати один і той самий код для створення різноманітних продуктів.
* Ізолює складний код конструювання продукту від його головної бізнес-логіки.

Недоліки:

* Ускладнює код програми за рахунок додаткових класів.
* Клієнт буде прив’язаний до конкретних класів будівельників, тому що в інтерфейсі будівельника може не бути методу отримання результату.

83. Подати за допомогою діаграми класів структуру патерна Оболонка (Wrapper), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування



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

## Застосування

* Якщо вам потрібно додавати об’єктам нові обов’язки «на льоту», непомітно для коду, який їх використовує.
* Об’єкти вкладаються в обгортки, які мають додаткові поведінки. Обгортки і самі об’єкти мають однаковий інтерфейс, тому клієнтам не важливо, з чим працювати — зі звичайним об’єктом чи з загорнутим.
* Якщо не можна розширити обов’язки об’єкта за допомогою спадкування.
* У багатьох мовах програмування є ключове слово final, яке може заблокувати спадкування класу. Розширити такі класи можна тільки за допомогою Декоратора.

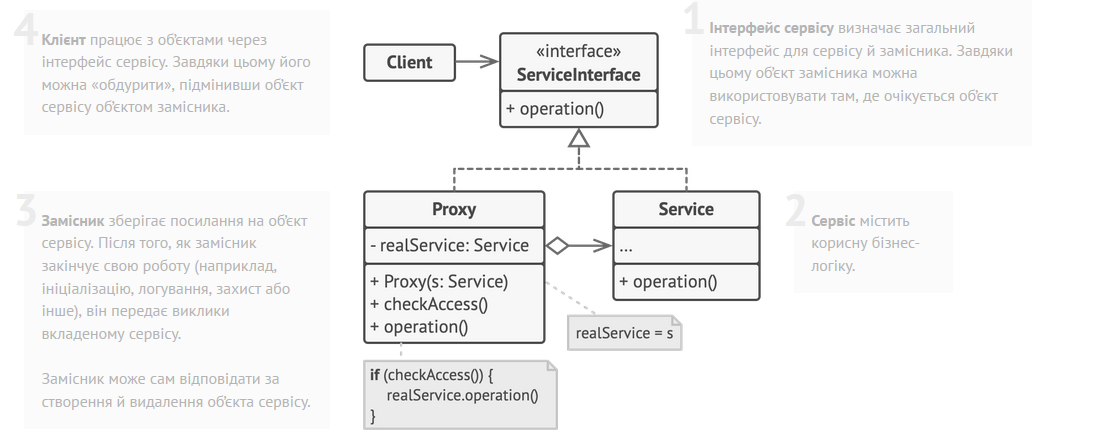
Переваги:

* Більша гнучкість, ніж у спадкування.
* Дозволяє додавати обов’язки «на льоту».
* Можна додавати кілька нових обов’язків одразу.
* Дозволяє мати кілька дрібних об’єктів, замість одного об’єкта «на всі випадки життя».

Недоліки:

* Важко конфігурувати об’єкти, які загорнуто в декілька обгорток одночасно.
* Велика кількість крихітних класів.

84. Подати за допомогою діаграми класів структуру патерна Сурогат (Surrogate), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування



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

Застосування

* Лінива ініціалізація (віртуальний проксі). Коли у вас є важкий об’єкт, який завантажує дані з файлової системи або бази даних.
* Замість того, щоб завантажувати дані відразу після старту програми, можна заощадити ресурси й створити об’єкт тоді, коли він дійсно знадобиться.
* Захист доступу (захищаючий проксі). Коли в програмі є різні типи користувачів, і вам хочеться захистити об’єкт від неавторизованого доступу. Наприклад, якщо ваші об’єкти — це важлива частина операційної системи, а користувачі — сторонні програми (корисні чи шкідливі).
* Проксі може перевіряти доступ під час кожного виклику та передавати виконання службовому об’єкту, якщо доступ дозволено.
* Локальний запуск сервісу (віддалений проксі). Коли справжній сервісний об’єкт знаходиться на віддаленому сервері.

Переваги:

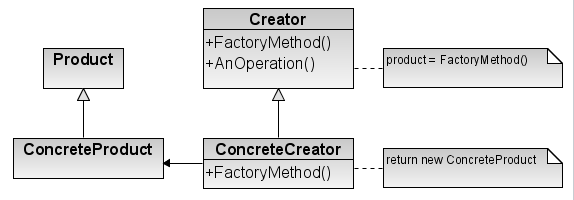
* Дозволяє контролювати сервісний об’єкт непомітно для клієнта.
* Може працювати, навіть якщо сервісний об’єкт ще не створено.
* Може контролювати життєвий цикл службового об’єкта.

Недоліки:

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

85. Подати за допомогою діаграми класів структуру патерна Фабрика (Factory), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

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



* **Product** — продукт:
  + визначає інтерфейс об'єктів, що створюються фабричним методом;
* **ConcreteProduct** — конкретний продукт:
  + реалізує інтерфейс *Product*;
* **Creator** — творець:
  + оголошує фабричний метод, що повертає об'єкт класу *Product*. *Creator* може також визначати реалізацію за умовчанням фабричного методу, що повертає об'єкт *ConcreteProduct*;
  + може викликати фабричний метод для створення об'єкта *Product*;
* **ConcreteCreator** — конкретний творець:
  + заміщує фабричний метод, що повертає об'єкт *ConcreteProduct*.

Застосування

* Коли типи і залежності об’єктів, з якими повинен працювати ваш код, невідомі заздалегідь.
* Коли ви хочете надати користувачам можливість розширювати частини вашого фреймворку чи бібліотеки.
* Коли ви хочете зекономити системні ресурси, повторно використовуючи вже створені об’єкти, замість породження нових.

Переваги:

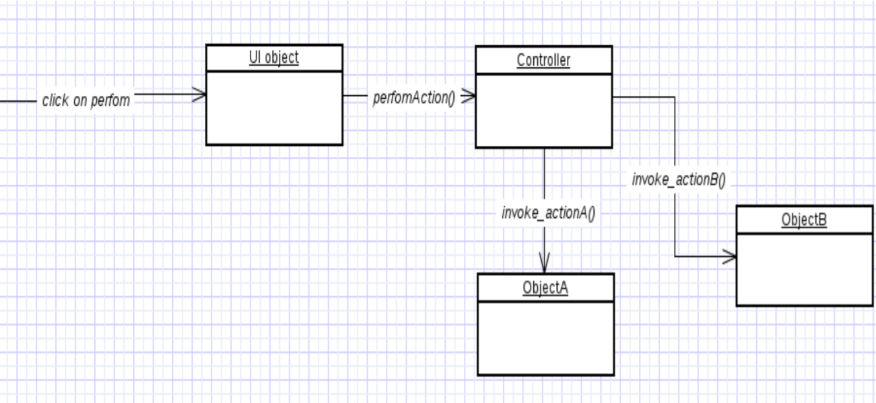
* Позбавляє клас від прив’язки до конкретних класів продуктів.
* Виділяє код виробництва продуктів в одне місце, спрощуючи підтримку коду.
* Спрощує додавання нових продуктів до програми.
* Реалізує принцип відкритості/закритості.

Недостатки:

* Може призвести до створення великих паралельних ієрархій класів, адже для кожного класу продукту потрібно створити власний підклас творця.

86. Подати за допомогою діаграми класів структуру патерна Контроллер (Controller), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

Контролер отримує введення користувача, обробляє його і посилає назад результат обробки, наприклад, у вигляді подання.



Переваги:

Зручно накопичувати інформацію про системні події (в разі, якщо системні операції виконуються в деякій певній послідовності).

Поліпшуються умови для повторного використання компонентів (системні події обробляються Контроллером а не елементами інтерфейсу користувача).

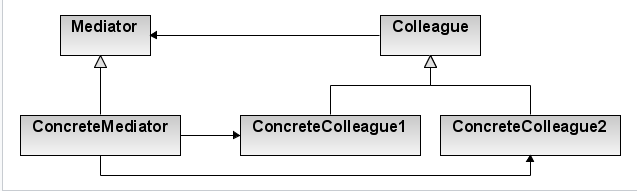
Недоліки:

Контролер може бути перевантажений.

87. Подати за допомогою діаграми класів структуру патерна Перенаправлення (Indirection), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

Перенаправлення – те саме, що й Посередник.

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



* **Mediator** – посередник:
  + визначає інтерфейс для обміну інформацією з об'єктами *Colleague*;
* **ConcreteMediator** – конкретний посередник:
  + реалізує кооперативну поведінку, координуючи дії об'єктів *Colleague*;
  + володіє інформацією про колег, та підраховує їх;
* **Класи Colleague** – колеги:
  + кожному класу *Colleague* відомо про свій об'єкт *Mediator*;
  + усі колеги обмінюються інформацією виключно через посередника, інакше за його відсутності їм довелося б спілкуватися між собою напряму

Слід використовувати шаблон *Посередник* у випадках, коли:

* існують об'єкти, зв'язки між котрими досить складні та чітко задані. Отримані при цьому залежності не структуровані та важкі для розуміння;
* не можна повторно використовувати об'єкт, оскільки він обмінюється інформацією з багатьма іншими об'єктами;
* поведінка, розподілена між кількома класами, повинна піддаватися налагодженню без створювання множини підкласів.

Переваги

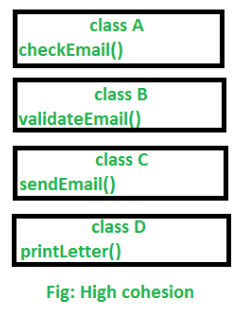
* Усуває залежності між компонентами, дозволяючи використовувати їх повторно.
* Спрощує взаємодію між компонентами.
* Централізує керування в одному місці.

Недоліки

* Посередник може **сильно «роздутися»**

88. Подати за допомогою діаграми класів структуру патерна Високе зчеплення (HighCohesion), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

Шаблон вирішує загальну проблему управління складністю за рахунок регулювання ступеня зачеплення класів. Зачеплення (cohesion) (або більш точно, функціональне зачеплення) - це міра пов'язаності і сфокусованість обов'язків класу. Вважається, що елемент має високий ступінь зачеплення, якщо його обов'язки тісно пов'язані між собою і він не виконує величезних обсягів роботи. Клас з низьким ступенем зачеплення виконує багато різнорідних функцій або незв'язаних між собою обов'язків.



Використання:

Необхідно забезпечити виконання об'єктами різнорідних функцій.

Переваги:

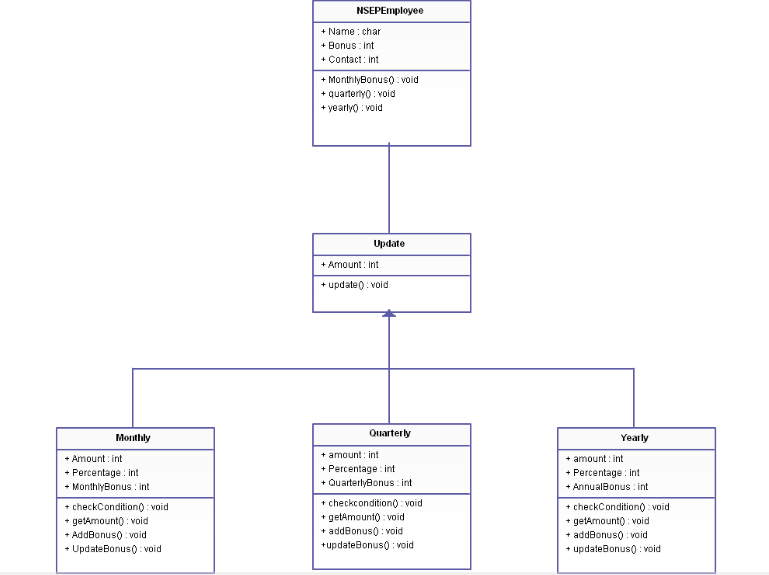
Класи з високим ступенем зачеплення прості в підтримці і повторному використанні.

Недоліки:  
Іноді буває невиправдано використовувати високу зачеплення для розподілених серверних об'єктів. В цьому випадку для забезпечення швидкодії необхідно створити кілька більших серверних об'єктів зі слабким зачепленням.

89. Подати за допомогою діаграми класів структуру патерна Транзакція (Transaction), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування

Організовує бізнес-логіку в процедури, які керують кожна своїм запитом. Більшість бізнес-додатків можна представити у вигляді набору транзакцій. Якісь із них вибирають дані, якісь - змінюють. Кожна взаємодія користувача і системи містить певний набір дій. У деяких випадках це може бути просто висновок даних з БД. В інших випадках ці дії який можуть містити в собі безліч обчислень і перевірок.

Патерн Transaction організовує всю цю логіку в одну процедуру, працюючи в БД безпосередньо або через тонку обгортку. Кожна транзакція має свій Transaction, хоча загальні підзадачі можуть бути розбиті на процедури.



Переваги:

Головним достоїнством типового рішення транзакції є простота. Саме такий вид організації логіки, ефективний з точки зору сприйняття і продуктивності, дуже характерний і природний для невеликих додатків.

Недоліки:

Повторення фрагментів коду. Оскільки кожен сценарій транзакції покликаний обслуговувати одну транзакцію, всі загальні порції коду доводиться відтворювати знову і знову.