Питання до іспиту АППЗ

Оглавление

[8. Проаналізувати співвідношення варіантів використання архітектури та детального проектування. (Лекція 6, Співвідношення варіантів використання, архітектури і детального проектування) 9](#_Toc43204997)

[16. Обґрунтувати використання детальних діаграм послідовності. (Лекція 7, діаграми послідовності) 15](#_Toc43204998)

[22. Визначити поняття інваріантів класів. (Лекція 8) 20](#_Toc43204999)

[23. Обґрунтувати використання компонентів при розробці архітектури програмного забезпечення. (Лекція 2) 21](#_Toc43205000)

[24. Проаналізувати інваріанти передумови та післяумови. (Лекція 8) 22](#_Toc43205001)

[25. Дати оцінку використання моделей при розробці архітектури програмного забезпечення. (Лекція 2) 23](#_Toc43205002)

[26. Визначити прийоми специфікації алгоритмів. (Лекція 8) 26](#_Toc43205003)

[27. Дати оцінку перевагам та недолікам використання програмної архітектури заснованої на потоках даних. (Лекція 3) 27](#_Toc43205004)

[36. Проаналізувати використання креаційних зразків у детальному проектуванні. 33](#_Toc43205005)

[38. Проаналізувати використання структурних зразків у детальному проектуванні. 35](#_Toc43205006)

[39. Описати інструментальні засоби для розробки програмної архітектури. 37](#_Toc43205007)

[40. Проаналізувати використання поведінкових зразків у детальному проектуванні. 39](#_Toc43205008)

[41. Описати етапи контролю якості при виборі архітектури. 41](#_Toc43205009)

[42. Визначити поняття зразків проектування заснованих на поведінці додатків. 43](#_Toc43205010)

[43. Дати оцінку перевірці архітектури за допомогою варіантів використання. (Лекція 5, Перевірка архітектури за допомогою варіантів використання) 45](#_Toc43205011)

[45. Описати етапи інспектування вибору архітектури. (Лекція 5, Інспектування вибору архітектури) 46](#_Toc43205012)

[47. Перерахуйте основні стандарти для документування проекту програмного продукту. (Лекція 4, Єдина система програмної документації) 47](#_Toc43205013)

[48. Проаналізувати співвідношення варіантів використання архітектури та детального проектування. (Лекція 6, Співвідношення варіантів використання, архітектури і детального проектування) 48](#_Toc43205014)

[49. Навести переваги використання рівневих архітектур. (Лекція 3, Рівнева архітектура, в лекції є ще більше діаграм та тексту) 50](#_Toc43205015)

[52. Обґрунтувати використання блок-схем у детальному проектуванні. (Лекція 8) 54](#_Toc43205016)

[57. Навести класифікацію архітектур програмного забезпечення. 61](#_Toc43205017)

[58. Проаналізувати використання інтерфейсів у детальному проектуванні. 63](#_Toc43205018)

[59. Дати оцінку використання каркасів у проектуванні програмного забезпечення 64](#_Toc43205019)

[60. Обґрунтувати використання детальних діаграм потоків даних. 70](#_Toc43205020)

[61. Подати за допомогою діаграми класів структуру патерна Спостерігач (Observer), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 71](#_Toc43205021)

[62. Подати за допомогою діаграми класів структуру патерна Команда (Command), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 73](#_Toc43205022)

[63. Подати за допомогою діаграми класів структуру патерна Ітератор (Iterator), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 76](#_Toc43205023)

[64. Подати за допомогою діаграми класів структуру патерна Пристосуванець (Flyweight), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 78](#_Toc43205024)

[65. Подати за допомогою діаграми класів структуру патерна Міст (Bridge), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 81](#_Toc43205025)

[66. Подати за допомогою діаграми класів структуру патерна Замісник (Proxy), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 84](#_Toc43205026)

[67. Подати за допомогою діаграми класів структуру патерна Декоратор (Decorator), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 86](#_Toc43205027)

[68. Подати за допомогою діаграми класів структуру патерна Адаптер (Adapter), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 88](#_Toc43205028)

[69. Подати за допомогою діаграми класів структуру патерна Інформаційний експерт (Information Expert) , коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 91](#_Toc43205029)

[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 93](#_Toc43205030)

[78. Подати за допомогою діаграми класів структуру патерна Зберігач (Memento), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 102](#_Toc43205031)

[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 103](#_Toc43205032)

[80. Подати за допомогою діаграми класів структуру патерна Шаблонний метод (Template Method), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 105](#_Toc43205033)

[81. Подати за допомогою діаграми класів структуру патерна Прототип (Prototype), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 106](#_Toc43205034)

[82. Подати за допомогою діаграми класів структуру патерна Будівельник (Builder), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 108](#_Toc43205035)

[83. Подати за допомогою діаграми класів структуру патерна Оболонка (Wrapper), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 110](#_Toc43205036)

[84. Подати за допомогою діаграми класів структуру патерна Сурогат (Surrogate), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 111](#_Toc43205037)

[85. Подати за допомогою діаграми класів структуру патерна Фабрика (Factory), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 112](#_Toc43205038)

[86. Подати за допомогою діаграми класів структуру патерна Контроллер (Controller), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 114](#_Toc43205039)

[87. Подати за допомогою діаграми класів структуру патерна Перенаправлення (Indirection), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 115](#_Toc43205040)

[88. Подати за допомогою діаграми класів структуру патерна Високе зчеплення (HighCohesion), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 117](#_Toc43205041)

[89. Подати за допомогою діаграми класів структуру патерна Транзакція (Transaction), коротко описати випадки його застосування, переваги та недоліки використання даного зразка проектування 118](#_Toc43205042)

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.

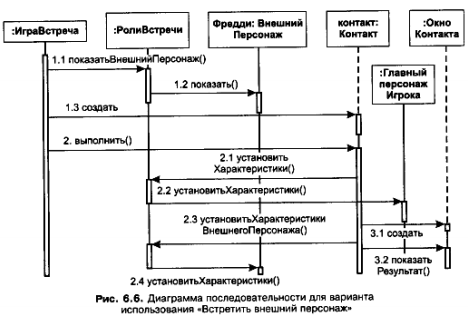
Нагадаємо, що варіанти використання можуть застосовуватися для представлення вимог, і тому ми застосовуємо їх для визначення ключових наслідуваних класів додатка. Тепер візьмемо діаграми послідовності, що побудовані на варіантах використання, і надамо відповідні класи з необхідними для виконання послідовностей 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. (Лекція 8)

Інваріант - це твердження, що залишається справедливим на всьому протязі обчислень. Добре продумані класи часто мають інваріанти.

Інваріанти класу є ефективними вимогами до об’єктів класу, виражені в термінах атрибутів класу. Ці вимоги можуть бути перенесені прямо на вимоги до застосування, і тому можуть увійти в SRS. Інакше вони пред'являються в ході проектування. Інваріанти класу набувають форми обмежень на значення. Як і С-вимоги, вони часто називаються правилами бізнесу.

Інваріант класу - інваріант, який використовується для обмеження об'єктів класу. Методи класу повинні зберігати інваріант.

Під час створення класів встановлюються їх інваріанти, які постійно підтримуються між викликами публічних методів. Тимчасове порушення класової інваріантності між приватними викликами методу можливо, хоча і небажано.

Інваріант об'єкта представляє собою конструкцію програмування, що складається з набору інваріантних властивостей. Це гарантує, що об'єкт завжди буде відповідати визначеним умовам, і тому методи можуть завжди посилатися на об'єкт без ризику зробити неточні презумпції. Визначення інваріантів класів може допомогти програмістам і тестувальникам виявити більше помилки при тестуванні програмного забезпечення

# 23. Обґрунтувати використання компонентів при розробці архітектури програмного забезпечення. (Лекція 2)

У другій половині 90-х років сильно зріс інтерес до поняття «компонент». Компонентами є повторно використовувані об'єкти, які не вимагають знання програмного забезпечення, що використовує їх. Наочним прикладом технології компонентів можуть служити об'єкти СОМ і Java Beans. Компоненти можуть бути об'єктами в звичайному розумінні об'єктно-орієнтованого програмування, але з невеликими поправками на забезпечення їх автономності. Одні компоненти використовуються іншими за допомогою агрегації і взаємодіють в основному за допомогою подій.

Зразки проектування можуть бути використані і на рівні каркаса, і усередині проектних класів на рівні архітектури, і на рівні детального проектування.

Зазвичай ніхто не намагається застосувати зразки проектування усередині класів наочної області, оскільки останні розроблені індивідуально і відповідають безпосередньо безлічі вимог. Як далі буде показано, зразки проектування часто вимагають введення класів, що не є класами наочної області, наприклад абстрактних класів.

# 24. Проаналізувати інваріанти передумови та післяумови. (Лекція 8)

Один із способів управління роботою функцій заснований на застосуванні інваріантів функцій. Так називається твердження про взаємозв'язки змінних, що гарантовано задовольняються функціями.

Прикладом з відеогри Зустріч може послужити наступний можливий інваріант для методу adjustQuality(): Сума значень характеристик персонажа постійна.

Іншими словами, коли значення однієї з характеристик міняється за допомогою методу adjustQuality(), значення інших характеристик змінюються так, що загальна сума значень усіх характеристик залишається незмінною.

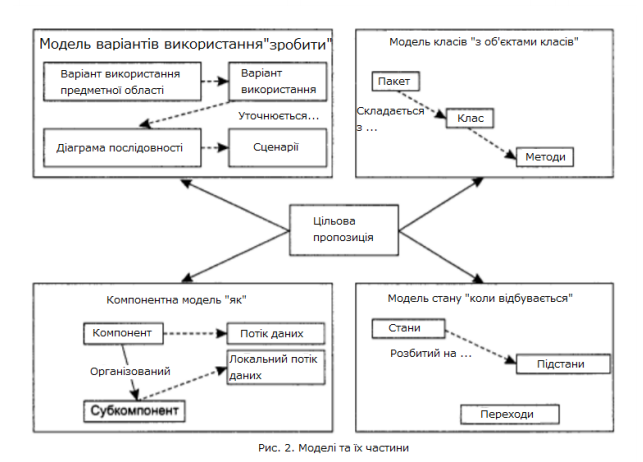
Ефективним способом визначення функцій є використання передумов і післяумов. Передумови описують співвідношення між змінними і константами, існування яких передбачається до моменту виконання функції. Післяумови описують ці співвідношення після виконання функції.

Кожен інваріант може бути замінений повторенням затвердження інваріанту одночасно в передумові та післяумові.

# 25. Дати оцінку використання моделей при розробці архітектури програмного забезпечення. (Лекція 2)

Зазвичай нам необхідно описати додаток з декількох точок зору. Це можна порівняти з архітектурою будинку, яка вимагає декількох проекцій, таких як план розташування ділянок землі, вертикальний і фронтальний види, план водопроводу і так далі. В світі програмування проекції називаються моделями. За останні роки в цій області з'явилася безліч прекрасних розробок. Різні моделі проектованого застосування показані на рис. 1 і 2.





*Модель варіантів використання* є колекцією варіантів використання. Вони пояснюють, що саме має робити додаток. Початкова версія варіантів використання підходить для використання як С-вимог (іноді їх називають «ділові варіанти використання» або «варіанти використання наочної області»), В процесі реалізації проекту вони набувають властивостей діаграм послідовності спеціального вигляду. Їх реалізують у вигляді конкретних сценаріїв, які потім використовуються при тестуванні. Як приклад варіанту використання для гри Зустріч можна узяти випадок зустрічі із зовнішнім персонажем, розглянутий в прикладі.

*Модель класів*. Ми вже розглянули достатньо багато моделей класів (діаграм класів). Модель класів пояснює побудова блоків, з яких буде сформовано застосування. Моделі класів часто називають об'єктними моделями. Усередині моделі класів ми можемо показати методи і атрибути.

*Модель компонентів* є колекцією діаграм потоків даних. Вони пояснюють, яким чином додаток працюватиме в термінах переміщення даних. Модель компонентів гри Зустріч у вправах включатиме діаграму потоків даних про характеристики між зовнішнім персонажем і персонажем гравця при їх зустрічі.

*Модель переходів* станів є колекцією діаграм переходів станів. Модель переходів станів визначає момент часу, в який додаток здійснює свою роботу. У грі Зустріч вже розроблено таку модель (розділ 3 авт.). Вона показує реакцію гри на появу інших персонажів, а також на запити користувача встановити характеристики, закрити вікно, вийти з гри і так далі

Усередині кожної моделі ми опрацьовуємо рівні деталізації, кількість яких росте. Залежно від об'ємів роботи ми можемо ітеративно застосовувати ділення на рівні деталізації усередині кожної з моделей. Архітектура застосування часто виражається в термінах однієї з моделей і підтримується рештою моделей. USDP включає модель реалізації. яка повинна бути виконана з урахуванням організації коду. У кожній архітектурі є щонайменше одна модель класів, здатна реалізувати цю архітектуру

# 26. Визначити прийоми специфікації алгоритмів. (Лекція 8)

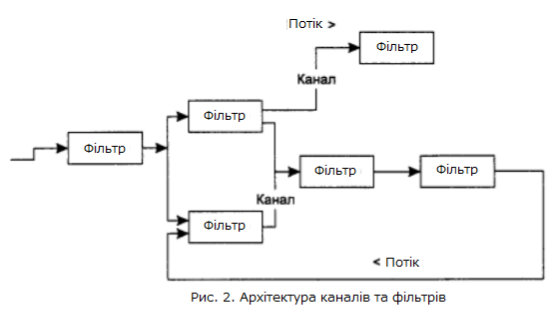
Після визначення функцій, які необхідно реалізувати, може виявитися корисним описати використовуваний алгоритм, коротко зупиняючись на вихідному коді. Основна перевага цього підходу полягає в тому. що розробник може проінспектувати алгоритм окремо без поглиблення в складності програмування. Завдяки цьому деякі дефекти можуть бути виявлені до того, як вони переростуть в дефекти програмного коду. Чим важливіше метод, тим важливіше ця діяльність. Методи зі складним розгалуженням є першими кандидатами для побудови блок-схем.

Одними з основних прийомів специфікації алгоритмів є блок-схеми та псевдокод.

# 27. Дати оцінку перевагам та недолікам використання програмної архітектури заснованої на потоках даних. (Лекція 3)

Для представлення деяких додатків якнайкраще підходять потоки даних між процесами обробки даних. Таке представлення ілюструють діаграми потоків дачних (DFD - Data Flow Diagram). Кожен процес обробки даних на діаграмі потоків даних проектується незалежно від інших. Дані приходять з різних джерел.

Архітектура потоків даних, що відноситься до архітектури типу канали і фільтри представлена на рис. 2. Така архітектура потоків даних СКЛАДАЄТЬСЯ з процесів (фільтрів), здатних у будь-який момент часу прийняти потоки як вхідну інформацію (послідовність даних однакового виду). Кожний фільтр повинен проектуватися незалежно від інших. Така архітектура може бути легко реалізована за допомогою каналів Unix.



Архітектура каналів і фільтрів має явну перевагу - модульність. Приклад додатку з такою архітектурою показаний на рис. 3. Цей додаток обслуговує транзакції рахунків, що приходять у випадкові моменти часу по лініях зв'язку. Архітектура містить крок для реєстрації транзакцій у разі відмови системи.



Протягом десятиліть потоки даних були найбільш загальним способом відображення архітектури, і можна з упевненістю сказати, що вони не втратять актуальності і в майбутньому. Для розробників цілком природньо уявляти собі, що дані подорожують від одного вузла до іншого і обробляються в кажному вузлі. Недоліком діаграм потоків даних є те, що їх відображення у програмному коді не цілком зрозуміла, причому незалежно від того, чи йде мова про об'єктно-орієнтований код, чи ні.

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

Архітектура незалежних компонентів складається з компонентів, що працюють паралельно (теоретично) і що час від часу взаємодіють один з одним.

Для того, щоб відобразити переваги та недоліки такої програмної архітектури розглянемо конкретні приклади. У відношенні клієнт-сервер серверні компоненти обслуговують потреби клієнтів за допомогою запитів. Клієнт-серверні відношення мають величезну перевагу – мале зчеплення між компонентами.

Компонент працює як сервер ефективніше, якщо він має вузький інтерфейс. "Вузький" означає, що інтерфейс (по суті, набір функцій) не має надмірностей, зібраний в одному місці і чітко визначений. Зразок проектування Facade надає пакетам класів саме такий інтерфейс. Це можна рахувати, як ще одна перевага. Адже чим вужчий інтерфейс у компонента тим ефективніше він працює і є більш зрозумілим.

Розглядаючи додаток зразка проектування Facade, слід враховувати додаткові витрати на забезпечення доступу користувачів пакету до методів. Наприклад, припустимо, що у нас є пакет Chemistry. Він містить клас Molecule, який у свою чергу містить корисний метод atomicWeight(). Якщо ми використовуємо facade-класс MoleculeFacade, то MoleculeFacade повинен містити відкритий метод getAtomicWeight(), що передає управління методу atomicWeightd(). Це вимагає додаткової роботи і узгодження під час процесу розробки, але витрати на це з лишком окупляться при зростанні модульності.

World Wide Web можна розглядати як ціле покоління клієнт-серверної архітектури, в якій архітектуру один сервер – десятки клієнтів замінила архітектура один сервер -мільйони клієнтів.

Архітектура незалежних елементів частенько складається з джерела даних і деякої кількості клієнтів. Дані клієнтів повинні оновлюватися кожного разу, коли в джерелі відбувається яка-небудь зміна.

Як архітектуру, яка здатна задовільняти дані вимоги, можна використовувати зразок проектування Observer. Сторони, яким необхідно оновлювати дані, називаються спостерігачами і є підкласами одного абстрактного класу, який ми назвемо Observer.

Перевага полягає в тому. що така модель дозволяє ліквідовувати спостерігачів і створювати нові, не порушуючи взаємодії інших спостерігачів. Недолік зразка проектування Observer проявляється у тому разі, коли дуже небагатьом із спостерігачів необхідно реагувати на зміни, що відбуваються: у такому разі численні сповіщення стають причиною даремної витрати ресурсів. Крім того, цю модель також небажано застосовувати, коли політика оновлення пропонується і визначається більше природньо самими спостерігачами, або коли вона значно відрізняється для різних спостерігачів

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

Архітектура віртуальних машин розглядає додаток як програму, що написана на спеціальній мові. Через те, що має бути реалізований інтерпретатор цієї мови, ця архітектура окупається, тільки якщо будуть написані декілька програм, що генерують додаток.

Перевага використання архітектури віртуальних машин в тому, що ви можете вільно генерувати додатки, виражаючи їх спеціальною мовою.

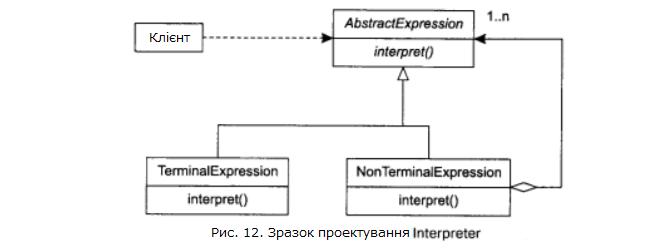
Архітектура віртуальних машин дуже корисна, якщо додаток складається з процесів із складними елементами (наприклад, замовлення), які можна виразити за допомогою граматики. Як додатковий приклад можна привести додаток, що виконує просте програмування на рівні користувача на спеціальній мові. Звичайний користувач може написати, наприклад, такий сценарій:

Перевірка балансу / додати суииу на рахунок + відняти дефіцит із залишку: Зберегти звіт / c : Reports + стандартні заголовки \* замінити "Ed" на "А1" Роздрукувати звіт / стандартні заголовки Відправити звіт за адресою Jayne0xyz.net

Архітектура віртуальних машин аналізує і інтерпретує такі сценарії.

Реалізація повної віртуальної машини вимагає побудови інтерпретатора. У загальному випадку інтерпретація викликається операцією – назвемо її interpret() – над програмою на нашій мові. Інтерпретація простого поодинокого элементу (наприклад, CPU) загалом проста (наприклад, це може означати вимогу вивести take CPU in the example). Проблема полягає в запуску Interpret) при застосуванні до складнішої програми.

Один з шляхів рішення – використовувати зразок проектування Interpreter. Гамма [33] вказує, що доцільно використовувати цей зразок проектування, коли граматика мала і швидкість не є важливим чинником. Зразок проектування Interpreter показаний на рис. 12.



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

Блок-схеми є одним із старих графічних представлень алгоритмів.

Наразі блок схеми використовуються менше через поширення об’єктно орієнтованого піходу. З даних тверджень ми можемо виявити таку перевагу, як зручне зображення алгоритмів лінійного та функціонального програмування. Недоліком є те, що в такому способі неможливо зобразити повний алгоритм складного, об’єктно орієнтованого додатку для максимального його розуміння.

Псевдокод - це варіант текстового представлення алгоритму, що дозволяє не вдаватися до деталей мов програмування.

Основною перевагою псевдокоду є його зрозумілість і можливість досить точно представити алгоритм. Друга перевага полягає в можливості інспектування алгоритмів на предмет їх коректності незалежно від нагромаджень мови програмування. Третя перевага в тому, що ми можемо накопичувати статистику по відсотку помилок в псевдокоді, а потім використовувати ці дані для передбачення кількості дефектів в кінцевому продукті. Багато організацій використовують інспектовані псевдокод та якість коментарів у лістингу вихідного програмного коду.

І псевдокод, і блок-схеми мають свої переваги і недоліки.

Переваги: Роблять алгоритм прозорішим. Підвищують дисципліну в процесі документування детального проектування. Дають додатковий рівень, на якому може бути проведено інспектування.

Недоліки: Створюють додатковий рівень документації, яку необхідно вести. Привносять можливість помилок при перекладі в програмний код. Може виникнути потреба в інструментах для витягання псевдокоду і спрощення малювання блок-схем.

33. Дати оцінку перевагам та недолікам використання репозиторної архітектури.

Архітектура, побудована головним чином навколо даних, називається репозиторною архітектурою. Більшість таких систем призначені для обробки транзакцій по відношенню до баз даних.

Слово "репозиторій" частенько використовується для вказівки на те, що додаток надає уніфікований інтерфейс доступу до колекції баз даних. Репозиторії не змінюють структури баз даних, а тільки надають уніфікований доступ до них. Цей особливий випадок репозиторних архітектур визначений Гарланом і Шоу.

Репозиторні архітектури використовуються у величезній кількості додатків, оскільки бази даних, які треба підтримувати, є присутні в багатьох архітектурах. Коли обробка трохи відрізняється від форматування даних з бази даних, репозиторна архітектура є самою відповідною. З іншого боку, наявність великої бази даних іноді свідчить про те, що значний об’єм обробки даних може впливати на архітектуру. У додатках, які спочатку не слід було проектувати за допомогою репозиторної моделі, швидко поширюється програмування ad hoc' (наприклад, з використанням процедур, що зберігаються).

# 36. Проаналізувати використання креаційних зразків у детальному проектуванні.

Зразок проектування - це знайдена на основі досвіду комбінація компонентів, зазвичай класів або об'єктів, яка вирішує певні загальні проектувальні завдання. Скористаємося аналогією з архітектурою будинку і розглянемо завдання проектування відокремленої будівлі на обширній території. Архітектура Ранчо (одноповерховий будинок) повністю задовольняє цим вимогам. Відмітьте, що Ранчо указує на загальну ідею проектування, що передбачає безліч реалізацій, і зовсім не є незмінним безліччю планів будинку

Креаційний зразок проектування пов'язаний із способами створення складних об'єктів, таких як лабіринти і дерева.

Креаційні зразки проектування дозволяють різноманітити створення об'єктів і керувати ними, у тому числі координувати групи об'єктів, створювані в ході роботи застосування. Від креаційних зразків проектування довелося б писати окремий програмний код для кожного типу об'єктів або групи об'єктів, які можуть знадобитися в ході роботи застосування. Креаційні зразки проектування зазвичай створюють об'єкти за допомогою окремого об'єкта або методу без прямого використання конструкторів.

Зразок проектування Singleton

Проблема: припустимо, що існує єдиний об'єкт, що належить класу С.

Зразок проектування Singleton : ідея цього зразка проектування реализуется завданням конструктора прихованого (чи, можливо, захищеного) класу. Це запобігає створенню об'єктів класу Із винятком випадку їх створення одним з методів самого класу С. Клас С - це задана частина статичних даних типу С, яка буде єдиною. Назвемо її TheCObject. Визначимо відкритий метод класу С, getTheObjectO. який повертатиме об'єкт TheCObject, якщо він вже існує, і створювати його інакше. Таким чином, для того, щоб отримати цей єдиний елемент класу, ми просто викликаємо метод C.getTheCObjectO. Статичний об'єкт класу С theCObject може ініціалізувати значення null, а потім встановлений в значення new С при першому ж виклику getCObjectO. У будь-якому випадку нульове значення theCObject може бути використане для перевірки існування цього об'єкту.

# 38. Проаналізувати використання структурних зразків у детальному проектуванні.

У структурних патернах розглядається питання про те, як з класів і об'єктів утворюються більш великі структури. Структурні патерни рівня класу використовують успадкування для складання композицій з інтерфейсів і реалізацій. Простий приклад - використання множинного спадкоємства для об'єднання декількох класів в один. В результаті виходить клас, що володіє властивостями всіх своїх батьків. Особливо корисний цей патерн, коли потрібно організувати спільну роботу декількох незалежно розроблених бібліотек. Інший приклад патерну рівня класу - адаптер. У загальному випадку адаптер робить інтерфейс одного класу (адаптируемого) сумісним з інтерфейсом іншого, забезпечуючи тим самим уніфіковану абстракцію різнорідних інтерфейсів. Це досягається за рахунок закритого успадкування адаптованості класу. Після цього адаптер висловлює свій інтерфейс в термінах операцій адаптованого класу. Замість композиції інтерфейсів або реалізацій структурні патерни рівня об'єкта компонують об'єкти для отримання нової функціональності. Додаткова гнучкість в цьому випадку пов'язана з можливістю змінити композицію об'єктів під час виконання, що неприпустимо для статичної композиції класів.

Прикладом структурного патерну рівня об'єктів є компоновщик. Він описує побудову ієрархії класів для двох видів об'єктів: примітивних і складових. Останні дозволяють створювати довільно складні структури з примітивних і інших складових об'єктів. У паттерне заступник об'єкт бере на себе функції іншого об'єкта. У заступника є багато застосувань. Він може діяти як локальний представник об'єкта, що знаходиться в віддаленому адресному просторі. Або представляти великий об'єкт, що завантажується на вимогу. Або обмежувати доступ до вирішального об'єкту. Заступник вводить додатковий непрямий рівень доступу до окремих властивостей об'єкта. Тому він може обмежувати, розширювати або змінювати ці властивості.

Патерн пристосуванець визначає структуру для спільного використання об'єктів. Власники поділяють об'єкти, щонайменше, з двох причин: для досягнення ефективності та несуперечності. Пристосуванець акцентує увагу на ефективності використання пам'яті. У додатках, в яких бере участь дуже багато об'єктів, повинні знижуватися накладні витрати на зберігання. Значної економії можна добитися за рахунок поділу об'єктів замість їх дублювання. Але об'єкт може бути розділяються, тільки якщо його стан не залежить від контексту. У об'єктів-пристосуванців такої залежності немає. Будь-яка додаткова інформація передається їм у міру необхідності. За відсутності контекстних залежностей об'єкти-пристосуванці можуть легко розділятися

# 39. Описати інструментальні засоби для розробки програмної архітектури.

Для полегшення процесу розробки програмного забезпечення використовується безліч автоматизованих інструментальних засобів. Деякі з них є колекцією класів з різними взаємозв'язками. Прикладами таких колекцій можуть служити Rational Rose or Rational Corporation і Together від Object International. Ці інструменти полегшують побудову об'єктних моделей, а також їх з'єднання з відповідним початковим кодом і діаграмами послідовності. Для вибирання інструментальних засобів моделювання складається список вимог до них. Цей процес аналогічний процесу аналізу вимог для розробки програмного додатку. Наведемо список деяких вимог до інструментів моделювання.

♦ [необхідно] Полегшення зображення об'єктних моделей і діаграм послідовності.

• Швидке створення класів.

• Легке редагування класів.

• Зміна масштабу зображення всередині частин моделі.

♦ [бажано] Можливість швидкого переходу від об'єктної моделі до вихідного коду.

♦ [необхідно] Повинен коштувати не більше $Х для одного користувача.

♦ [не обов’язково] Можливість зворотного проектування (тобто створення об'єктної моделі з початкового коду).

Пакети інструментальних засобів часто намагаються охопити і архітектуру, і детальне проектування, і реалізацію. Різні продавці розробляють системи з можливістю використання гіперпосилань між початковим кодом і документацією. Інструментальні засоби, орієнтовані на реалізацію, подібні Javadoc, можуть бути хорошим доповненням для процесу розробки. Клієнт-серверні інструменти, такі як Powerbuilder, цілком придатні для визначення архітектури, хоча вони визначають і реалізації. Javadoc дуже корисний при навігації по пакетах, оскільки він надає алфавітний список усіх класів, а також їх ієрархію

Інтерактивні середовища розробки (IDE) укомплектовані компіляторами і використовуються як інструменти часткового моделювання. Об’єктно-орієнтованні IDE в основному показують наслідування в ієрархічній формі. Цей факт притягає розробників із-за близькості цих інструментів до процесів компіляції і відладки. Проте IDE зазвичай мають недостатньо широкий спектр можливостей, щоб полегшити побудову архітектури і проектної роботи.

Інструментальні засоби зборки компонентів дозволяють створювати додатки за допомогою перетягування значків, що представляють елементи процесів. Середовища JavaBeans є прикладом таких інструментальних засобів. У таких середовищах об'єкти Java, класи яких відповідають стандарту Java Beans, можна узяти з бібліотек або створити самостійно і зв'язати через події. Стандарт Java Beans був створений для полегшення таких простих зборок за допомогою графічних інструментальних засобів.

Основна незручність при використанні інструментів моделювання пов'язана із залежністю проекту від третьої сторони - продавця. На додаток до усього, окрім складності самого додатку і проекту розробник повинен турбуватися про життєздатність продавця. Якщо продавець збанкрутить або оновлення для інструментальних засобів стане занадто дорогим - як це відіб'ється на проекті?

Але незважаючи на все це популярність інструментів моделювання росте по всьому світі. Тривалість їх використання обмежується продуктивністю і економічними чинниками.

# 40. Проаналізувати використання поведінкових зразків у детальному проектуванні.

Патерни поведінки пов'язані з алгоритмами і розподілом обов'язків між об'єктами. Мова в них йде не тільки про самі об'єкти і класах, але і про типові способах взаємодії. Патерни поведінки характеризують складний потік управління, який важко простежити під час виконання програми. Увага акцентована не на потоці управління як такому, а на зв'язках між об'єктами.

У патернах поведінки рівня класу використовується успадкування - щоб розподілити поведінку між різними класами. Більш простим і широко поширеним є шаблонний метод, який являє собою абстрактне визначення алгоритму. Алгоритм тут визначається покроково. На кожному кроці викликається або примітивна, або абстрактна операція. Алгоритм «обростає м'ясом» за рахунок підкласів, де визначені абстрактні операції. Інший патерн поведінки рівня класу - інтерпретатор, який представляє граматику мови у вигляді ієрархії класів і реалізує інтерпретатор як послідовність операцій над екземплярами цих класів

У патернах поведінки рівня об'єктів використовується не успадкування, а композиція. Деякі з них описують, як за допомогою кооперації безліч рівноправних об'єктів справляється із завданням, яка жодному з них не під силу. Важливо тут те, як об'єкти отримують інформацію про існування один одного. Об'єкти-колеги можуть зберігати посилання один на одного, але це збільшить ступінь пов'язаності системи. При максимальному ступені пов'язаності кожному об'єкту довелося б мати інформацію про всі інші. Цю проблему вирішує патерн посередник. Посередник, що знаходиться між об'єктами-колегами, забезпечує опосередкованість посилань, необхідну для розривання зайвих зв'язків.

Патерн ланцюжок обов'язків дозволяє і далі зменшувати ступінь пов'язаності. Він дає можливість посилати запити об'єкту не безпосередньо, а по ланцюжку «об'єктів-кандидатів». Запит може виконати будь-який «кандидат», якщо це допустимо в поточному стані виконання програми. число кандидатів заздалегідь не визначено, а підбирати учасників можна під час виконання. Патерн спостерігач визначає і відповідає за залежності між об'єктами. Класичний приклад спостерігача зустрічається в схемі модель / вид / контролер мови Smalltalk, де всі види моделі повідомляються про будь-які зміни її стану.

Інші патерни поведінки пов'язані з інкапсуляцією поведінки в об'єкті і делегуванням йому запитів. Патерн стратегія інкапсулює алгоритм об'єкта, спрощуючи його специфікацію і заміну. Патерн команда інкапсулює запит у вигляді об'єкта, який можна передавати як параметр, зберігати в списку історії або використовувати як-то інакше. Патерн стан інкапсудює стан об'єкта таким чином, що при зміні стану об'єкт може змінювати поведінка. Патерн відвідувач інкапсулює поведінку, яке в іншому випадку довелося б розподіляти між класами, а патерн ітератор абстрагує спосіб доступу і обходу об'єктів з деякого агрегату.

# 41. Описати етапи контролю якості при виборі архітектури.

Є велика різниця між тим, щоб просто запрограмувати деяку функцію і тим, щоб зробити її як якісний продукт. У першому випадку ми маємо код, про який не можна сказати нічого певного крім того, що він компылюється і працює. У другому випадку ми отримуємо код, який:

- задовольняє ясно сформульовані вимоги;

- перевіряє вхідні дані і передбачувано реагує на некоректні вхожние дані;

- ретельно проінспектований не тільки самим автором коду, але і іншими розробниками;

- пройшов вичерпне багатостороннє тестування;

- ретельно документований;

- має надійну оцінку ступеня дефектності (відсотка помилок)

Різниця тут приблизно того ж порядку, що і між виготовленням саморобної полички для книг і виготовленням балки моста, спроектованої з урахуванням можливих навантажень транспортного і пішохідного руху.

Аналогічно, високоякісний програмний продукт зазвичай є:

- розширюваним (готовим до можливих змін для розширення функціональності);

- готовим до змін (легко адаптованим до зміни вимог);

- стерпним (придатним до використання на декількох платформах);

- загальним (застосовним до декількох різних ситуацій).

Нашою метою є розробка стандартів для даних характеристик і створення продуктів, які задовольняли б цим специфікованим стандартам. Для цього ми повинні навчитися вимірювати якість, чисельно задавати необхідні рівні якості і вміти контролювати процес досягнення заданого рівня якості.

Процес контролю якості

Відповідальність за якість артефакту в першу чергу несе людина, що створює цей артефакт. Але, як би там не було, один в полі не воїн. Всім нам потрібний сторонній погляд на виконувану нами роботу. Цей погляд необхідний, щоб уникнути недалекоглядності, нереалістичною самооцінки і застою. Процес контролю якості являється також суспільним обов'язком частини роботи. Кожна частина роботи, виконана інженером, повинна бути детально перевірена принаймні однією людиною бажано незалежним від автора роботи.

На додаток до відповідальності індивідуальних розробників і перевірок роботи їхніх колег багато організацій визначили процес роздільної систематичної і повної перевірки, в подальшому - контроль якості (QA - QUALITY ASSURANCE).

У функції контролю якості входять перевірки, інспектування та тестування. Контроль якості повинен починатися разом з запуском кожного проекту. Найкраще залучати контроль якості і для перевірки правильності використовуваного процесу і актуальності документації.

Представник групи контролю якості часто бере участь в інспектуванні. В ідеалі контроль якості повинен виконуватися деякою незалежної організацією. Багато компаній занадто малі для здійснення такого складного контролю якості, і в цьому випадку самі інженери виконують функції контролю якості по відношенню до роботи один одного.

# 42. Визначити поняття зразків проектування заснованих на поведінці додатків.

Патерни поведінки пов'язані з алгоритмами і розподілом обов'язків між об'єктами. Мова в них йде не тільки про самі об'єкти і класах, але і про типові способах взаємодії. Патерни поведінки характеризують складний потік управління, який важко простежити під час виконання програми. Увага акцентована не на потоці управління як такому, а на зв'язках між об'єктами.

У патернах поведінки рівня класу використовується успадкування - щоб розподілити поведінку між різними класами. Більш простим і широко поширеним є шаблонний метод, який являє собою абстрактне визначення алгоритму. Алгоритм тут визначається покроково. На кожному кроці викликається або примітивна, або абстрактна операція. Алгоритм «обростає м'ясом» за рахунок підкласів, де визначені абстрактні операції. Інший патерн поведінки рівня класу - інтерпретатор, який представляє граматику мови у вигляді ієрархії класів і реалізує інтерпретатор як послідовність операцій над екземплярами цих класів

У патернах поведінки рівня об'єктів використовується не успадкування, а композиція. Деякі з них описують, як за допомогою кооперації безліч рівноправних об'єктів справляється із завданням, яка жодному з них не під силу. Важливо тут те, як об'єкти отримують інформацію про існування один одного. Об'єкти-колеги можуть зберігати посилання один на одного, але це збільшить ступінь пов'язаності системи. При максимальному ступені пов'язаності кожному об'єкту довелося б мати інформацію про всі інші. Цю проблему вирішує патерн посередник. Посередник, що знаходиться між об'єктами-колегами, забезпечує опосередкованість посилань, необхідну для розривання зайвих зв'язків.

Патерн ланцюжок обов'язків дозволяє і далі зменшувати ступінь пов'язаності. Він дає можливість посилати запити об'єкту не безпосередньо, а по ланцюжку «об'єктів-кандидатів». Запит може виконати будь-який «кандидат», якщо це допустимо в поточному стані виконання програми. число кандидатів заздалегідь не визначено, а підбирати учасників можна під час виконання. Патерн спостерігач визначає і відповідає за залежності між об'єктами. Класичний приклад спостерігача зустрічається в схемі модель / вид / контролер мови Smalltalk, де всі види моделі повідомляються про будь-які зміни її стану.

Інші патерни поведінки пов'язані з інкапсуляцією поведінки в об'єкті і делегуванням йому запитів. Патерн стратегія інкапсулює алгоритм об'єкта, спрощуючи його специфікацію і заміну. Патерн команда інкапсулює запит у вигляді об'єкта, який можна передавати як параметр, зберігати в списку історії або використовувати як-то інакше. Патерн стан інкапсудює стан об'єкта таким чином, що при зміні стану об'єкт може змінювати поведінка. Патерн відвідувач інкапсулює поведінку, яке в іншому випадку довелося б розподіляти між класами, а патерн ітератор абстрагує спосіб доступу і обходу об'єктів з деякого агрегату.

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

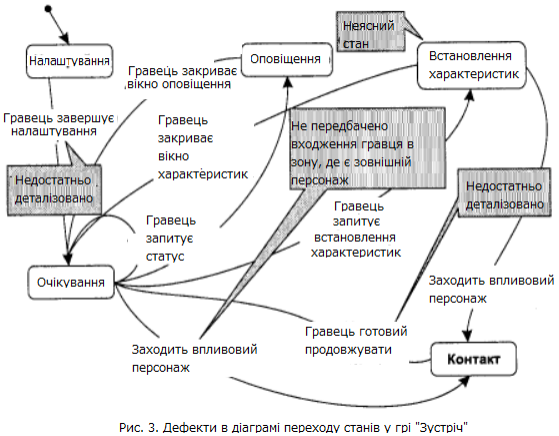
Варіанти використання отримують із вимог замовника. Тому варіанти використання не можуть враховувати архітектуру додатку по простій причині – вона ще не визначена. Після вибору архітектури корисно повернутися до розгляду варіантів використання і перевірити, чи адекватно їх підтримує архітектура. Оскільки 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).

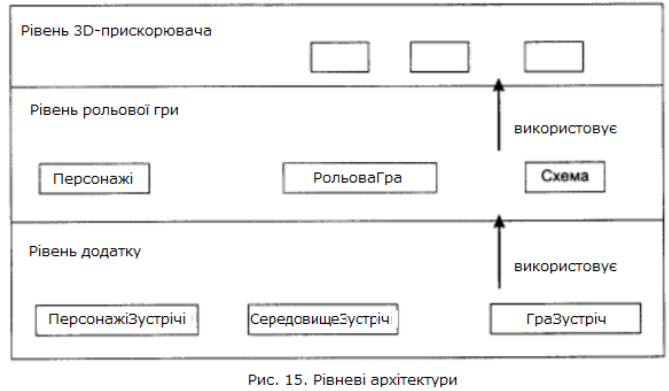
У проектуванні програмного забезпечення кожен наступний рівень накопичує додаткові класи. На кроці 1 варіанти використання фіксуются як частина вимог. На кроці 2 вони разом з іншими джерелами використовуються для визначення класів предметної області. На кроці 3, ми розробляємо програмну архітектуру (див. 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).

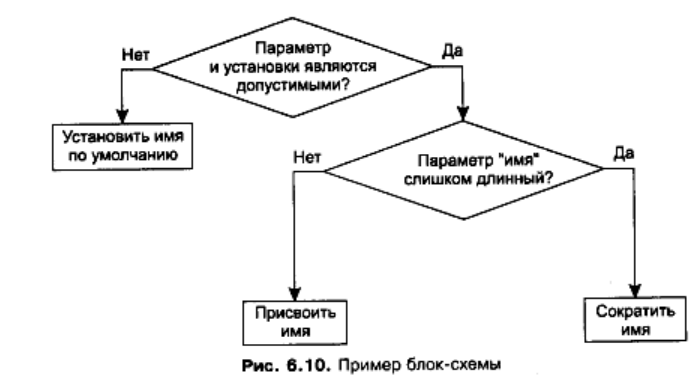
Рівнева архітектура має безліч переваг при повторному використанні. Бібліотека класів Java є дуже ефективною рівневою системою (пакет applet - рівень - заснований на пакеті awt, який у свою чергу заснований на пакеті lang. і т. д.)

50. Обґрунтувати використання детальних діаграм послідовності.

51. Визначити поняття створення архітектури програмного забезпечення.

# 52. Обґрунтувати використання блок-схем у детальному проектуванні. (Лекція 8)

Блок-схеми є одним із старих графічних представлень алгоритмів. Приклад блок-схеми для методу setNane() класу GameCharacter (ПерсонажГри), що демонструє два прості елементи блок-схем - умови (ромби) і дія (прямокутники). – приведений на рис. 6.10. Завдяки інспектуванню блок-схеми на рис. 6.10 ми можемо побачити, що перевірка допустимості імені параметра підкоряється досить туманній логіці.



Увесь доступний інструментарій для створення блок-схем з початковою коду - це один з видів зворотного проектування, при якому створюються артефакти, що інспектуються, проте допускається інспекція до програмування, що і є основною ідеєю використання блок-схем.

Використання блок-схем знизилося за два останні десятиліття двадцятого століття. Частково це сталося через поширення об’єктноорієнтованого підходу, який став досить популярний саме в ті роки у зв'язку з тенденцією скорочення кількості розгалужень завдяки розподілу функціональності по класах і використанні віртуальних функцій.

І псевдокод, і блок-схеми мають свої переваги і недоліки. Перерахуємо їх переваги.

♦ Роблять алгоритм прозорішим.

♦ Підвищують дисципліну в процесі документування детального проектування.

♦ Дають додатковий рівень, на якому може бути проведено інспектування:

⎫ допомагають відстежити дефекти до того, як вони з'являються в програмному коді;

⎫ підвищують надійність продукту.

♦ Часто знижують загальну вартість розробки. Проте вони мають наступні недоліки.

♦ Створюють додатковий рівень документації, яку необхідно вести.

♦ Привносять можливість помилок при перекладі в програмний код.

♦ Може виникнути потреба в інструментах для витягання псевдокоду і спрощення малювання блок-схем.

Рішення про використання псевдокоду і (або) блок-схем приймається в залежності від кожного конкретного додатку. Деякі розробники уникають використання блок-схем, вважаючи їх застарілим засобом. Проте блок-схеми і псевдокод можуть виявитися дуже корисними для окремої частини застосувань, де вони сприятимуть виробництву продуктів більш високої якості.

53. Описати процедуру вибору архітектури.

Більшість застосувань можуть бути реалізовані за допомогою різних архітектур. Деякі варіанти можуть бути набагато краще за інших. Такі важливі рішення, як вибір архітектури, не приймаються без первинної розробки і порівняння альтернатив. Запропонована архітектура ретельно аналізуються, оскільки усунення дефектів на ранній стадії коштує набагато менше, ніж їх виправлення під час реалізації проекту. Один із способів вибору – присвоїти вагу необхідним характеристикам і призначити нечіткий коефіцієнт якості для кожного кандидата

Для конкретного проекту розробки програмного забезпечення може бути декілька відповідної архітектури, з якої необхідно вибрати кращу. Зазвичай буває складно задовольнити усі вимоги, оскільки архітектура може виконувати одну з вимог і не виконувати іншу. З цієї причини усім вимогам необхідно присвоїти пріоритети. Наведемо приклад списку основних цілей розробки.

- Розширення.

Полегшення додавання нових властивостей.

- Зміни

Полегшення зміни вимозі

- Простота:

- простота розуміння;

- простота реалізації.

- Ефективність:

- досягнення високої швидкості : виконання і (чи) компіляції;

- досягнення малого розміру : об'єктного коду і (чи) початкового кілка.

Розширення визначає міру, в якій архітектура повинна підтримувати додавання нових можливостей в застосування. Найчастіше чим краще архітектура пристосована до розширення, тим більше складну структуру вона має і більше часу вимагається на розробку. Розширюваність зазвичай вимагає введення вищих абстракції в процес. Одним з важливих завдань при виборі міри універсальності є визначення класу можливих розширень. Ми не можемо проектувати в розрахунок на усі можливі розширення. У зв'язку з цим дуже корисні необов'язкові і бажані вимоги, оскільки вони показують, в яку сторону буде спрямований розвиток застосування.

ОДИН ІЗ СПОСОБІВ ПОЧАТИ ВИБІР БАЗОВОЇ АРХІТЕКТУРИ

1. Розробити ментальну модель застосування на високому рівні, начебто це було маленьке застосування. Наприклад, персональне фінансове

застосування отримує або видає гроші у будь-якому порядку під управлінням інтерфейсу користувача.

2. Виконати декомпозицію на необхідні компоненти. Пошук високої зв'язності і низ-кого зчеплення. Зокрема, для персонального фінансового застосування виконується декомпозиція на Активи. Постачальники і Інтерфейс.

3. Повторити цей процес для компонентів.

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.

Вибір архітектури визначає спосіб реалізації вимог на високому рівні абстракції. Саме архітектура майже повністю визначає такі характеристики ПЗ як надійність, переносимість і зручність супроводу. Архітектура значно впливає і на зручність використання і ефективність ПЗ, які визначаються також і реалізацією окремих компонентів. Значно менше вплив архітектури на функціональність – зазвичай для реалізації заданої функціональності можна використовувати різні архітектури. Тому вибір між тією або іншою архітектурою визначається насамперед саме нефункціональними вимогами і необхідними властивостями ПЗ в аспектах зручності супроводу та переносимості. При цьому для побудови гарної архітектури треба враховувати можливі протиріччя між вимогами до різних характеристик і вміти вибирати компромісні рішення, що дають прийнятні значення за всіма показниками.

По-перше, для підвищення ефективності в загальному випадку вигідніше використовувати монолітні архітектури, в яких виділено невелике число компонентів (або єдиний компонент) – цим забезпечується економія як пам'яті, оскільки кожен компонент зазвичай має свої дані, а тут число компонентів мінімально, так і часу роботи, оскільки можливість оптимізувати роботу алгоритмів обробки даних є також, зазвичай, тільки в рамках одного компонента.

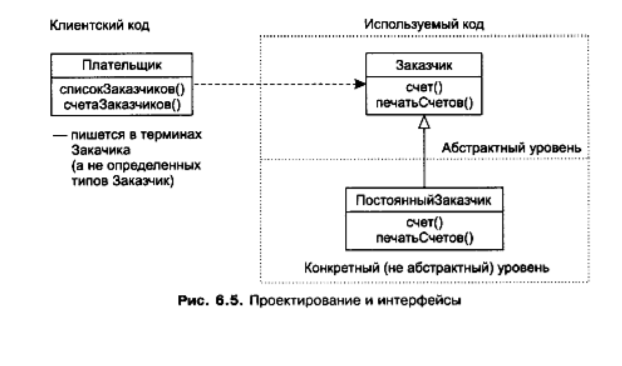
З другого боку, для підвищення зручності супроводу, навпаки, краще розбивати систему на велике число окремих компонентів, з тим, щоб кожен з них вирішував свою невелику, але чітко визначену частину загальної задачі. При цьому, якщо виникають зміни у вимогах або проекті, їх зазвичай можна звести до зміни однієї-кількох таких підзадач, і, відповідно, змінювати тільки ті компоненти, що відповідають за вирішення цих підзадач.

З третього боку, для підвищення надійності краще використовувати дублювання функцій, тобто зробити кілька компонентів відповідальними за вирішення однієї підзадачі. Причому, оскільки помилки в ПЗ найчастіше носять невипадковий характер (тобто вони повторювані, на відміну від апаратного забезпечення, де помилки пов'язані насамперед з випадковими змінами характеристик середовища і можуть бути подолані простим дублюванням компонентів, без зміни їх внутрішньої реалізації), краще використовувати різноматітні способи вирішення однієї і тієї ж задачі в різних компонентах.

Список стандартів, що регламентують опис архітектури та проектну документацію взагалі, виглядає так: − IEEE 1016-1998 Recommended Practice for Software Design Descriptions; − IEEE 1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems.

58. Проаналізувати використання інтерфейсів у детальному проектуванні.

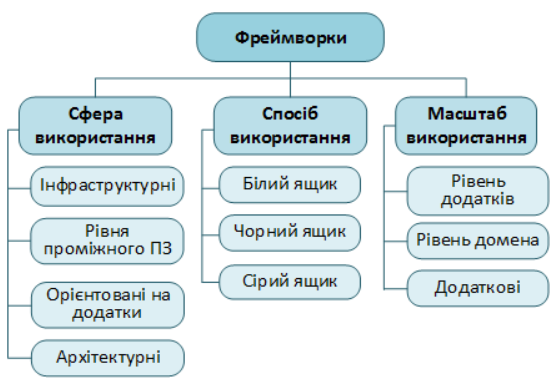
Проектування з використанням інтерфейсів можна порівняти із заключенням контракту. Елемент додатку (наприклад, клас Замовник), забезпечує необхідну функціональність, гарантує надання функцій з певними іменами, типами параметрів і значень, що повертаються (наприклад, void рахунок( void ) і boolean печатьСчетов( String типСчета )). Елементи додатку, що використовують цю основу функціональності, слід проектувати так, щоб вони «не знали» подробиць реалізації функціональності. Застосування інтерфейсів в проектуванні може мати різні форми. Одна з них – це абстракція. Наприклад, якщо ми пишемо код для об'єктів Ссавці, ми повинні робити це так, щоб згадувати тільки об'єкти Тварини. Іншими словами, ми намагаємося використовувати тільки інтерфейс Тварини. Це підвищує гнучкість і застосовність нашого проектування Розглянемо ще один приклад. Допустимо, ми пишемо код застосування для роботи із замовниками. Це можна розуміти як написання коду на основі інтерфейсу Замовник. Взагалі кажучи, ми можемо розглядати абстрактний клас Замовник з неабстрактним підкласом, наприклад ПостійнийЗамовник (рис. 6.5). Подібний підхід до проектування гнучкіший, ніж робота з конкретним (не абстрактним) класом Замовник, так він дозволяє нам легко додавати нові типи замовників з їх індивідуальними версіями методу рахунок(), і при цьому нам не треба міняти код, що маніпулює об'єктами Замовник. Таке ділення на абстрактний і конкретний рівні характерно для багатьох зразків проектування.



59. Дати оцінку використання каркасів у проектуванні програмного забезпечення

**Інтернет:**

Термін фреймворк можна визначити як загальноприйняті архітектурно-структурні рішення й підходи до проектування. Можна сказати, що фреймворк являє собою загальне рішення складного завдання.



Інфраструктурні фреймворки (System Іnfrastructure Frameworks) спрощують процес розробки інфраструктурних елементів, застосовуються усередині організації й не продаються.

Фреймворки рівня проміжного програмного забезпечення (Mіddleware Frameworks) застосовуються для вбудовування додатків і компонентів.

Фреймворки, орієнтовані на додатки, використовуються для підтримки систем, орієнтованих на роботу з кінцевими користувачами в конкретній предметній області.

У відповідності зі стандартом КОЛЕС 42010 архітектурний фреймворк визначається як «сукупність угод, принципів і практик, які використовуються для опису архітектур і прийнятих відповідно деякому предметному домену й (або) у співтоваристві фахівців (зацікавлених осіб)». Архітектурний фреймворк містить у собі опис зацікавлених осіб, типові проблеми предметної області, архітектурні точки зору й методи їхньої інтеграції.

Фреймворки, використовувані за принципом білого ящика (Archіtecture-drіveng framework), застосовують методи спадкування й динамічного зв’язування для формування основних елементів додатка. Такі фреймворки визначаються через інтерфейси об’єктів, що додають у систему. Для роботи з ними необхідна докладна інформація про класи, розширення яких необхідно.

Фреймворки, що функціонують за принципом чорного ящика, також називають фреймворками, керованими даними. Основними механізмами формування додатків, у цьому випадку, виступають композиція й параметризація, при цьому функціональність забезпечується додаванням додаткових компонентів. Слід зазначити, що процес використання фреймворків, що працюють за принципом чорного ящика простіше, ніж працюючих за принципом білого ящика, однак їхня розробка складніше.

На практиці застосовують підхід сірого ящика (grey box), що є комбінацією обох підходів.

Фреймворки рівня додатків (applіcatіon frameworks) надають функціонал по реалізації типових додатків (GUІ, бази даних і т.д).

Фреймворки рівня домену (Domaіn Frameworks) застосовуються для створення додатків у певній предметній області. Їхня класифікація представлена на рис. 2.10.

М’які фреймворки мають на увазі можливість власного настроювання для вирішення конкретного завдання, у той час як тверді – ні.

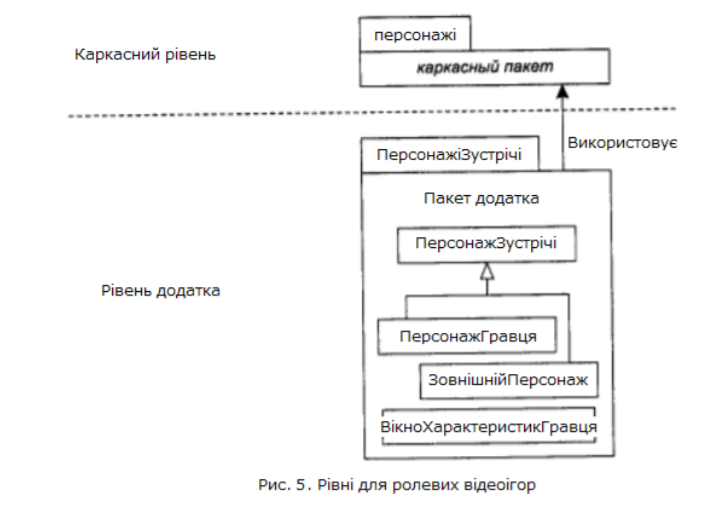
Допоміжні фреймворки (Support Frameworks) застосовуються для рішення приватних завдань.

У світі існує величезна кількість різних фреймворків. Як приклади можна виділити п’ять найбільш відомих:

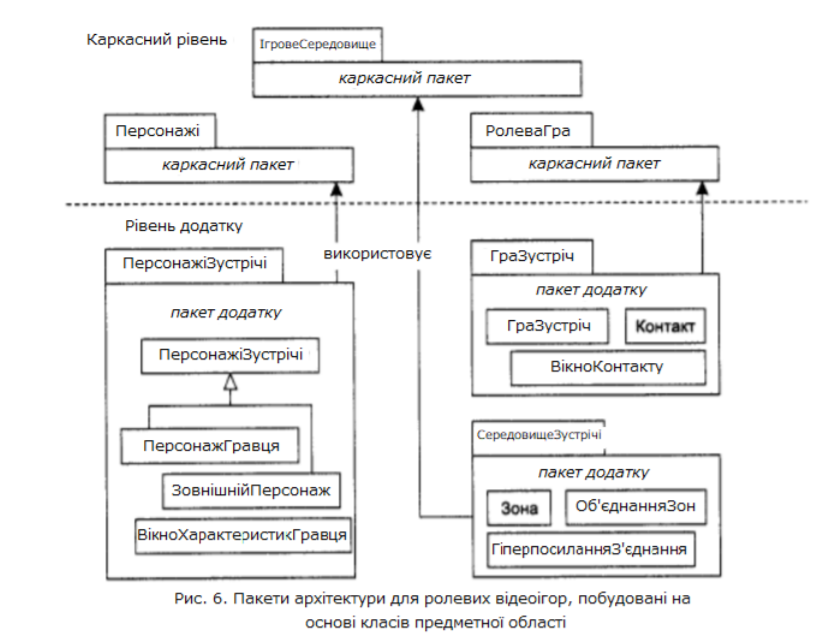
**Лекція:**

Каркас — це колекція класів, використовуваних в декількох різних додатках. Часто класи усередині каркаса взаємозв'язані. Вони можуть бути абстрактними і використовуватися через спадкоємство. Інтерфейс прикладного програмування (API), реалізований в Java, є прикладом корисних каркасних пакетів. Він показав, наскільки співтовариство розробників потребує багатої колекції каркасів для виконання своєї роботи. Базові пакети Java API можуть бути використані у величезній кількості різноманітних додатків. Пов'язання пакетів застосувань з каркасними пакетами здійснюється за допомогою агрегації і (або) наслідування. Розглянемо, наприклад, використання пакету Java Windowing Toolkit {awt): замість того щоб модифікувати awt, ми створюємо класи графічного інтерфейсу користувача, які успадковані від awt або агрегують об'єкти awt як атрибути. Існує думка, що каркаси слід створювати тільки в тому випадку, якщо вони, так само як і Java API. використовуватимуться великою кількістю додатків. Проте розробка часткового каркаса паралельно із додатком дає безліч переваг, навіть коли немає упевненості в тому, що цей каркас буде придатний для великого числа додатків. Цей частковий каркас нерідко служить незмінним абстрактним рівнем, який успадковують класи багатьох застосувань.

Як приклад створимо каркас, який можна використовувати в грі Зустріч. Виконаємо декомпозицію гри як застосування. Варіанти декомпозиції легко отримати, групуючи класи наочної області, отримані при аналізі вимог (Зона, Персонажвстречи, Ігравстреча, Контакт, Окноконтакта. Внешнійперсонаж, Персонажігрока і Окнохарактерістікигрока). Кожен з цих класів повинен підходити для пакету додатка, і кожен пакет додатка повинен використовувати один або більше за пакетів каркаса. Наприклад, згідно цього принципу, один каркасний пакет Персонаж може визначати участь героїв в грі Зустріч. Також може бути створений пакет додатка Персонажвстречи, використовуючий Персонаж і що містить класи для персонажів класу Ігравстреча (рис. 5). Декомпозиція ролевої гри на пакети каркаса і пакети застосування Зустріч показана на рис. 6.

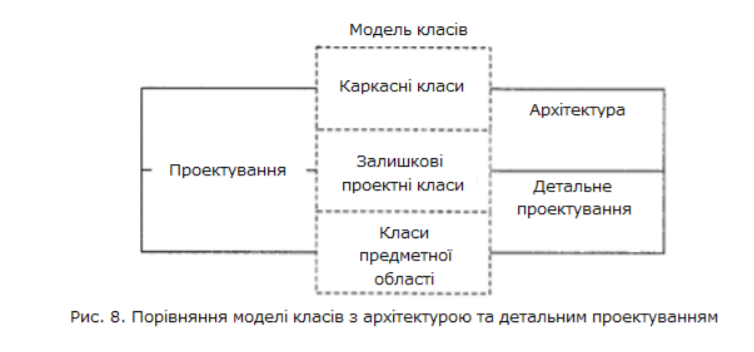


Каркасний пакет Персонажі визначає і персонажів, що керуються гравцем і персонажів, що керуються додатками. Каркасний пакет Артефакти складається з елементів, що не належать ніяким іншим пакетам. Необхідність пакету Артефакти гри на цій стадії ще не зовсім ясна. Можливо, ще рано міркувати про те, чи повинні. наприклад, аксесуари розглядатися як частина рівня або щити міститися в пакеті Персонаж.



Ми прагнули отримати високу зв'язність і низьке зчеплення в додатку Зустріч, групуючи логічну суть (персонажі, гра, управління і відображення). Якби ми групували класи так, як вони з'являються на моніторі (персонажі усередині зони), то пакети мали б неприйнятно низьку зв'язність, оскільки вони не є сильно зв'язаними класами. Зчеплення в цьому випадку виявилася б дуже високою, оскільки ми повинні були б забезпечити доступ і до зони, і до персонажа (а це дуже різна суть) ззовні пакету. Шлях, який приводить нас до моделі класів, представлений на рис. 7. Класи наочної області отримані як результат аналізу вимог. Класи каркаса отримані з розробки архітектури для застосувань, подібних тому, яке ми проектуємо. Таким чином, класи каркаса або є частиною раніше існуючого пакету, або розроблені, як в процесі аналізу архітектури. Нарешті, класи (проектні класи), що залишилися, додані для завершення проектування. Завершене проектування складається зі всіх класів наочної області, всіх класів проектування і деяких каркасних класів (рис. 7). (Зазвичай ми не використовуємо всі каркасні класи.) Каркасні класи, використані в проекті, є частиною архітектури застосування. Всі класи наочної області — це частина детального проектування, оскільки вони специфічні для кожного застосування, отримані з вимог і не відносяться до архітектури. Як приклад візьмемо клас Персонаж Гравця з вправи. По суті, він є окремим класом, що описує поведінку персонажа. Деякі з проектних класів створюються в процесі проектування архітектури, інші - в процесі детального проектування.





60. Обґрунтувати використання детальних діаграм потоків даних.

Для детального проектування моделі даних повинні детально описуватися, а потім відображатися на функції або класи і методи. Дії, необхідні для опису усіх деталей прикладу банківської діаграми потоків даних з теми 3, демонструє рис. 6.8. Кожен елемент процесу розкривається в детальнішу діаграму, і цей процес розкриття триває, поки не буде досягнутий самий нижній рівень елементів процесу. На цьому рівні елементами, як правило, є окремі функції, можливо навіть з різних класів. Наприклад, функція getDeposit() розкривається в три функції (отримання пароля, його перевірка і вивід на екран). Дві з цих функцій взаємодіють з сховищами даних (локальним звітом транзакцій, списком неприпустимих користувачів і шаблоном для виводу на екран), які не були видні на діаграмі верхнього рівня. Відмітимо, що вхідні і вихідні дані з детальних розкриттів співпадають по версіях з тими даними, з яких вони були витягнуті. Не для усіх додатків можна отримати користь з діаграми потоків даних. Наприклад, в них немає особливої необхідності в нашому прикладі з грою Зустріч.



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

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

*Структура:*



1. Видавець володіє внутрішнім станом, зміни якого цікаво відслідковувати підписникам. Видавець містить механізм підписки: список підписників та методи підписки/відписки.
2. Коли внутрішній стан видавця змінюється, він сповіщає своїх підписників. Для цього видавець проходиться за списком підписників і викликає їхній метод сповіщення, який описаний в загальному інтерфейсі підписників.
3. Підписник визначає інтерфейс, яким користується видавець для надсилання сповіщень. Здебільшого для цього досить одного методу.
4. Конкретні підписники виконують щось у відповідь на сповіщення, яке надійшло від видавця. Ці класи мають дотримуватися загального інтерфейсу, щоб видавець не залежав від конкретних класів підписників.
5. Після отримання сповіщення підписнику необхідно отримати оновлений стан видавця. Видавець може передати цей стан через параметри методу сповіщення. Більш гнучкий варіант — передавати через параметри весь об’єкт видавця, щоб підписник міг сам отримати необхідні дані. Як варіант, підписник може постійно зберігати посилання на об’єкт видавця, переданий йому через конструктор.
6. Клієнт створює об’єкти видавців і підписників, а потім реєструє підписників на оновлення у видавцях.

***Випадки застосування:***

**Якщо після зміни стану одного об’єкта потрібно щось зробити в інших, але ви не знаєте наперед, які саме об’єкти мають відреагувати.**

 Описана проблема може виникнути при розробленні бібліотек користувацього інтерфейсу, якщо вам необхідно надати можливість стороннім класам реагувати на кліки по кнопках.

Патерн Спостерігач надає змогу будь-якому об’єкту з інтерфейсом підписника зареєструватися для отримання сповіщень про події, що трапляються в об’єктах-видавцях.

**Якщо одні об’єкти мають спостерігати за іншими, але тільки у визначених випадках.**

 Видавці ведуть динамічні списки. Усі спостерігачі можуть підписуватися або відписуватися від отримання сповіщень безпосередньо під час виконання програми.

*Переваги:*

* Видавці не залежать від конкретних класів підписників і навпаки.
* Ви можете підписувати і відписувати одержувачів «на льоту».
* Реалізує *принцип відкритості/закритості*.

*Недоліки:*

 Підписники сповіщуються у випадковій послідовності

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

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

*Структура:*



1. **Відправник** зберігає посилання на об’єкт команди та звертається до нього, коли потрібно виконати якусь дію. Відправник працює з командами тільки через їхній загальний інтерфейс. Він не знає, яку конкретно команду використовує, оскільки отримує готовий об’єкт команди від клієнта.
2. **Команда** описує інтерфейс, спільний для всіх конкретних команд. Зазвичай тут описується лише один метод запуску команди.
3. **Конкретні команди** реалізують різні запити, дотримуючись загального інтерфейсу команд. Як правило, команда не робить всю роботу самостійно, а лише передає виклик одержувачу, яким виступає один з об’єктів бізнес-логіки.

Параметри, з якими команда звертається до одержувача, необхідно зберігати у вигляді полів. У більшості випадків об’єкти команд можна зробити незмінними, передаючи у них всі необхідні параметри тільки через конструктор.

1. **Одержувач** містить бізнес-логіку програми. У цій ролі може виступати практично будь-який об’єкт. Зазвичай, команди перенаправляють виклики одержувачам, але іноді, щоб спростити програму, ви можете позбутися від одержувачів, «зливши» їхній код у класи команд.
2. **Клієнт** створює об’єкти конкретних команд, передаючи до них усі необхідні параметри, серед яких можуть бути і посилання на об’єкти одержувачів. Після цього клієнт зв’язує об’єкти відправників зі створеними командами.

*Застосування*

**Якщо ви хочете параметризувати об’єкти виконуваною дією.**

 Команда перетворює операції на об’єкти, а об’єкти, у свою чергу, можна передавати, зберігати та взаємозаміняти всередині інших об’єктів.

Скажімо, ви розробляєте бібліотеки графічного меню і хочете, щоб користувачі могли використовувати меню в різних програмах, не змінюючи кожного разу код ваших класів. Застосувавши патерн, користувачам не доведеться змінювати класи меню, замість цього вони будуть конфігурувати об’єкти меню різними командами.

**Якщо ви хочете поставити операції в чергу, виконувати їх за розкладом або передавати мережею.**

 Як і будь-які інші об’єкти, команди можна серіалізувати, тобто перетворити на рядок, щоб потім зберегти у файл або базу даних. Потім в будь-який зручний момент його можна дістати назад, знову перетворити на об’єкт команди та виконати. Так само команди можна передавати мережею, логувати або виконувати на віддаленому сервері.

**Якщо вам потрібна операція скасування.**

 Головна річ, яка потрібна для того, щоб мати можливість скасовувати операції — це зберігання історії. Серед багатьох способів реалізації цієї можливості патерн Команда є, мабуть, найпопулярнішим.

Історія команд виглядає як стек, до якого потрапляють усі виконані об’єкти команд. Кожна команда перед виконанням операції зберігає поточний стан об’єкта, з яким вона працюватиме. Після виконання операції копія команди потрапляє до стеку історії, продовжуючи нести у собі збережений стан об’єкта. Якщо знадобиться скасування, програма візьме останню команду з історії та відновить збережений у ній стан.

Цей спосіб має дві особливості. По-перше, точний стан об’єктів не дуже просто зберегти, адже його частина може бути приватною. Вирішити це можна за допомогою патерна [**Знімок**](https://refactoring.guru/uk/design-patterns/memento).

По-друге, копії стану можуть займати досить багато оперативної пам’яті. Тому іноді можна вдатися до альтернативної реалізації, тобто замість відновлення старого стану, команда виконає зворотню дію. Недолік цього способу у складності (іноді неможливості) реалізації зворотньої дії.

*Переваги:*

* Прибирає пряму залежність між об’єктами, що викликають операції, та об’єктами, які їх безпосередньо виконують.
* Дозволяє реалізувати просте скасування і повтор операцій.
* Дозволяє реалізувати відкладений запуск операцій.
* Дозволяє збирати складні команди з простих.
* Реалізує *принцип відкритості/закритості*.

*Недоліки:*

Ускладнює код програми внаслідок введення великої кількості додаткових класів.

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

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

*Структура:*



1. **Ітератор** описує інтерфейс для доступу та обходу елементів колекцій.
2. **Конкретний ітератор** реалізує алгоритм обходу якоїсь конкретної колекції. Об’єкт ітератора повинен сам відстежувати поточну позицію при обході колекції, щоб окремі ітератори могли обходити одну і ту саму колекцію незалежно.
3. **Колекція** описує інтерфейс отримання ітератора з колекції. Як ми вже говорили, колекції не завжди є списком. Це може бути і база даних, і віддалене API, і навіть дерево [Компонувальника](https://refactoring.guru/uk/design-patterns/composite). Тому сама колекція може створювати ітератори, оскільки вона знає, які саме ітератори здатні з нею працювати.
4. **Конкретна колекція** повертає новий екземпляр певного конкретного ітератора, зв’язавши його з поточним об’єктом колекції. Зверніть увагу на те, що сигнатура методу повертає інтерфейс ітератора. Це дозволяє клієнтові не залежати від конкретних класів ітераторів.
5. **Клієнт** працює з усіма об’єктами через інтерфейси колекції та ітератора. Через це клієнтський код не залежить від конкретних класів, що дозволяє застосовувати різні ітератори, не змінюючи існуючого коду програми.

*Застосування*

**Якщо у вас є складна структура даних, і ви хочете приховати від клієнта деталі її реалізації (з питань складності або безпеки).**

 Ітератор надає клієнтові лише кілька простих методів перебору елементів колекції. Це не тільки спрощує доступ до колекції, але й захищає її від необережних або злочинних дій.

**Якщо вам потрібно мати кілька варіантів обходу однієї і тієї самої структури даних.**

 Нетривіальні алгоритми обходу структури даних можуть мати досить об’ємний код. Цей код буде захаращувати все навкруги — чи то самий клас колекції, чи частина бізнес-логіки програми. Застосувавши ітератор, ви можете виділити код обходу структури даних в окремий клас, спростивши підтримку решти коду.

**Якщо вам хочеться мати єдиний інтерфейс обходу різних структур даних.**

 Ітератор дозволяє винести реалізації різних варіантів обходу в підкласи. Це дозволить легко взаємозаміняти об’єкти ітераторів в залежності від того, з якою структурою даних доводиться працювати.

*Переваги*

* Спрощує класи зберігання даних.
* Дозволяє реалізувати різні способи обходу структури даних.
* Дозволяє одночасно переміщуватися структурою даних у різних напрямках.

*Недоліки*

* Невиправданий, якщо можна обійтися простим циклом.

# 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:

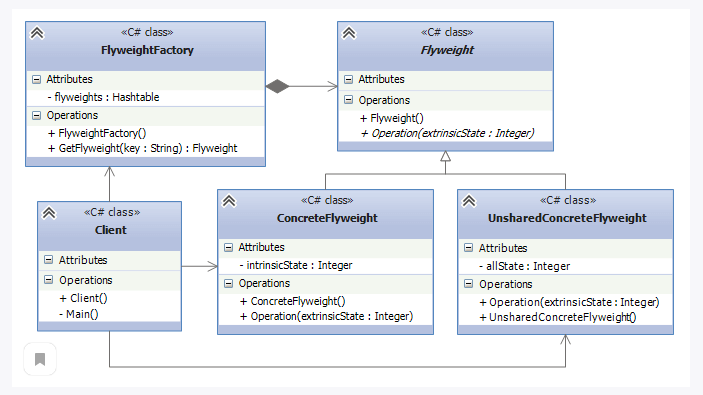
- Коли додаток використовує велику кількість одноманітних об'єктів, через що відбувається виділення великої кількості пам'яті

- 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;

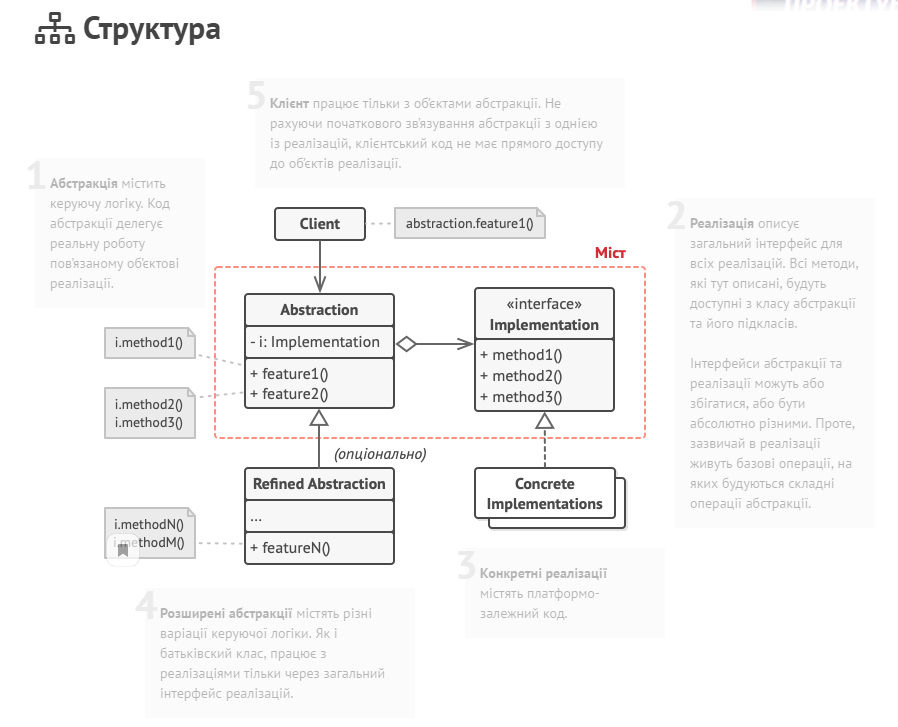
1. через це високі витрати оперативної пам’яті;
2. більшу частину стану об’єктів можна винести за межі їхніх класів;
3. великі групи об’єктів можна замінити невеликою кількістю об’єктів, що розділяються, оскільки зовнішній стан винесено.

# 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.

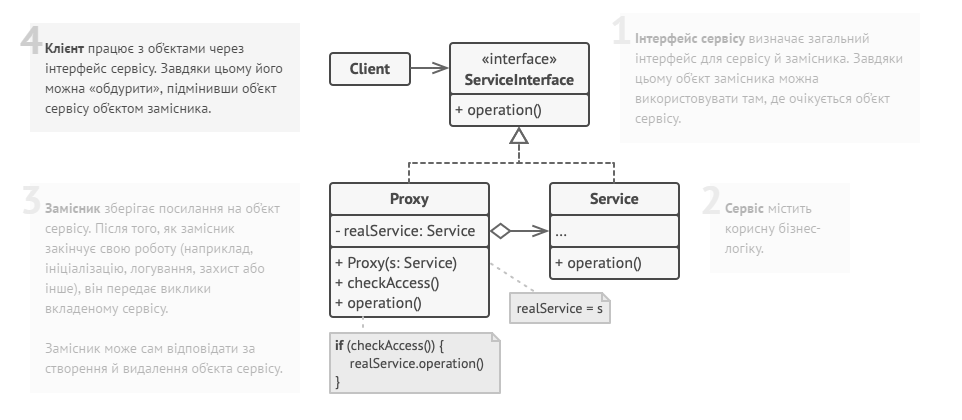
1. Якщо клас потрібно розширювати в двох незалежних площинах.

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.

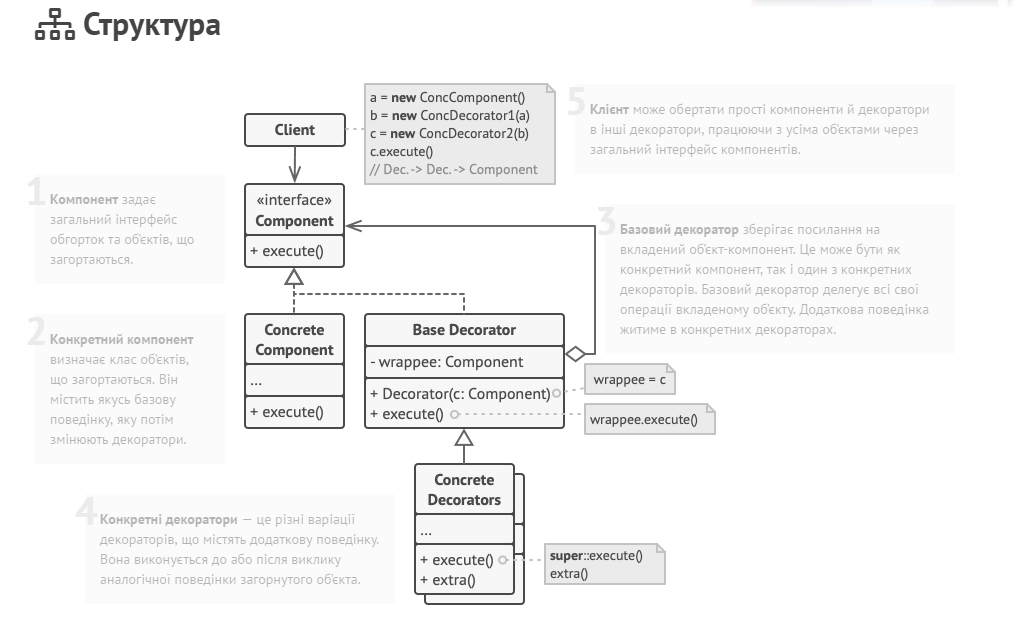
Замісник може підраховувати кількість посилань на сервісний об’єкт, які були віддані клієнту та залишаються активними. Коли всі посилання звільняться, можна буде звільнити і сам сервісний об’єкт (наприклад, закрити підключення до бази даних).

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.

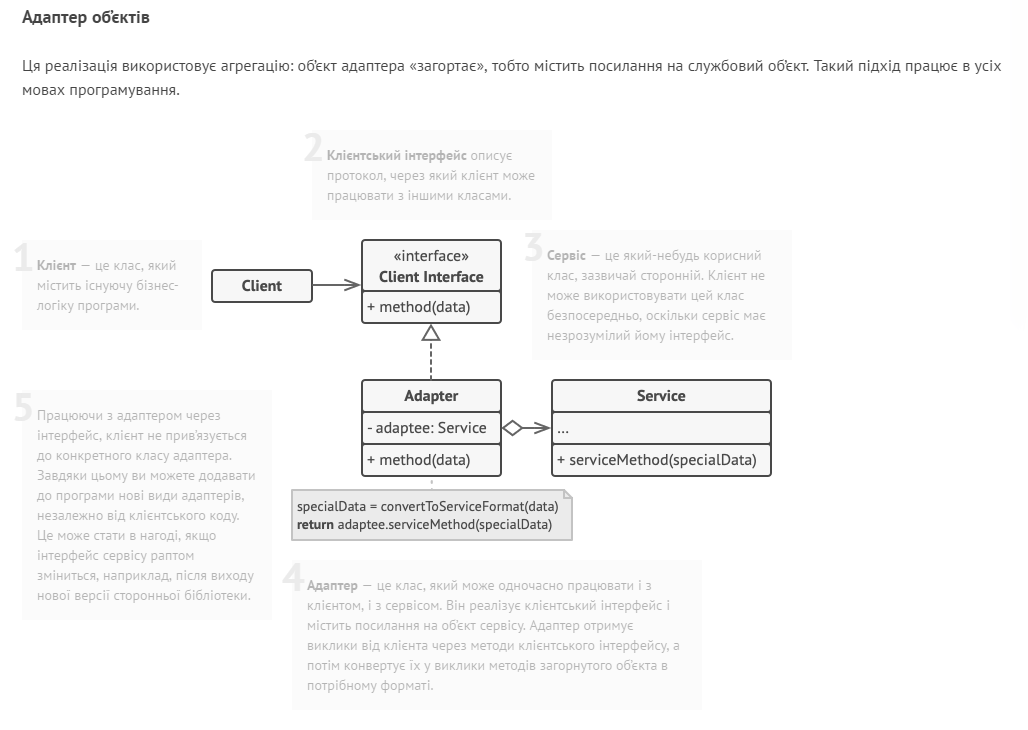
Адаптер загортає один з об’єктів так, що інший об’єкт навіть не підозрює про існування першого. Наприклад, об’єкт, що працює в метричній системі вимірювання, можна «обгорнути» адаптером, який буде конвертувати дані у фути.

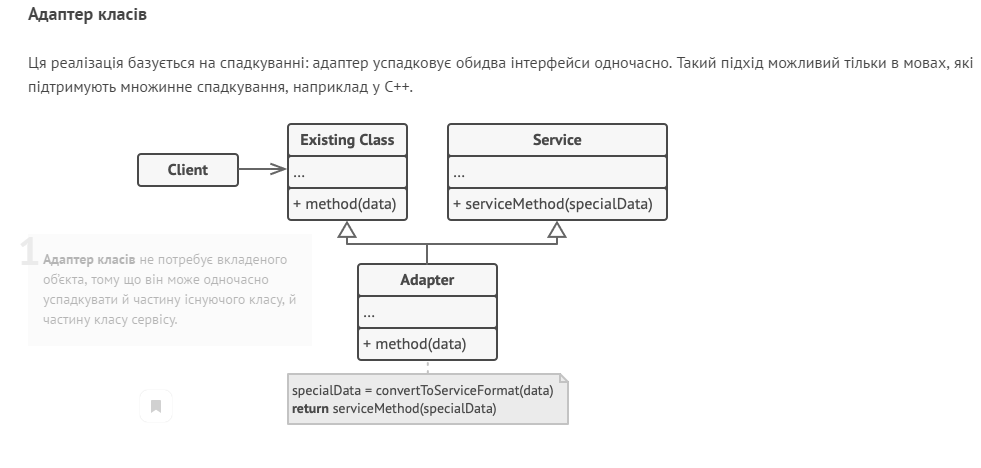
Адаптери можуть не тільки конвертувати дані з одного формату в іншій, але й допомагати об’єктам із різними інтерфейсами працювати разом. Це виглядає так:

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

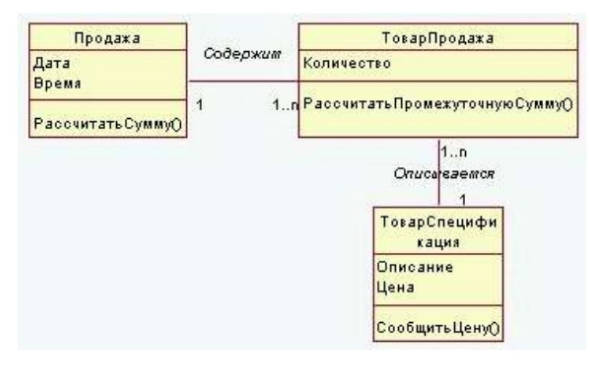
GRASP (General Responsibility Assignment Software Patterns) – загальні зразки розподілу обов'язків GRASP-патерни, що використовуються в об'єктно-орієнтованому проектуванні для вирішення спільних завдань за призначенням обов'язків класам і об'єктам.

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.

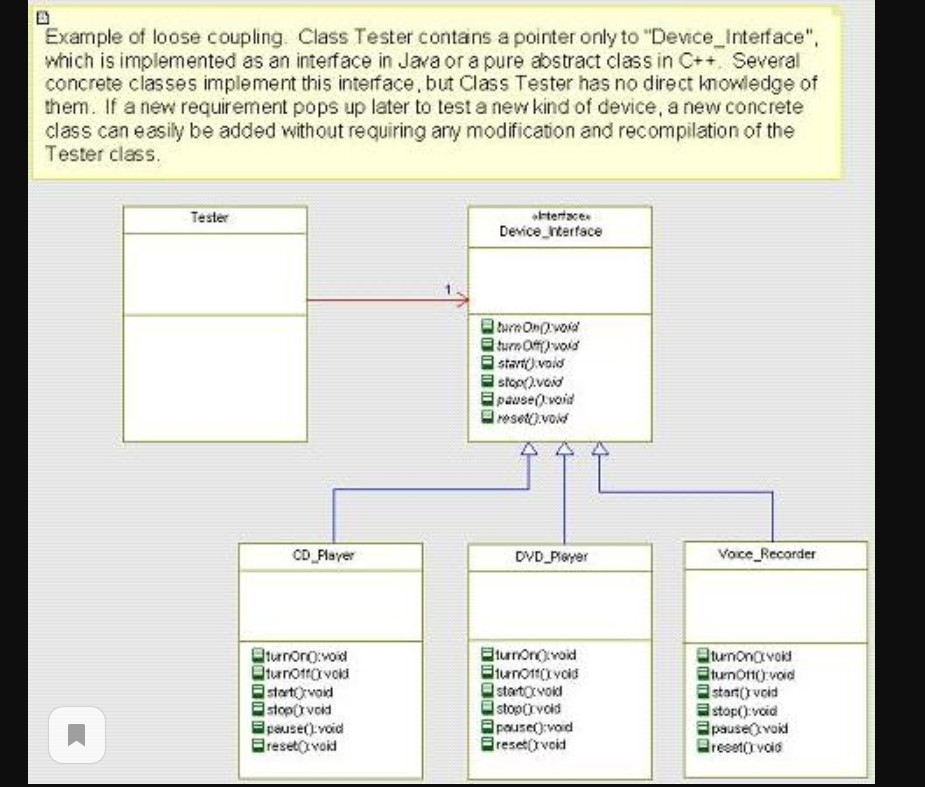
Прикладом гарного дизайну системи може служити набір утиліт GNU Binutils для Linux. В якому, кожна утиліта (якщо її розглядати як модуль) виконує лише мінімальні обов'язки (високе зачеплення) і майже нічого не знає про природу інших утиліт (низька зв'язність), в зв'язку з чим може бути легко замінена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

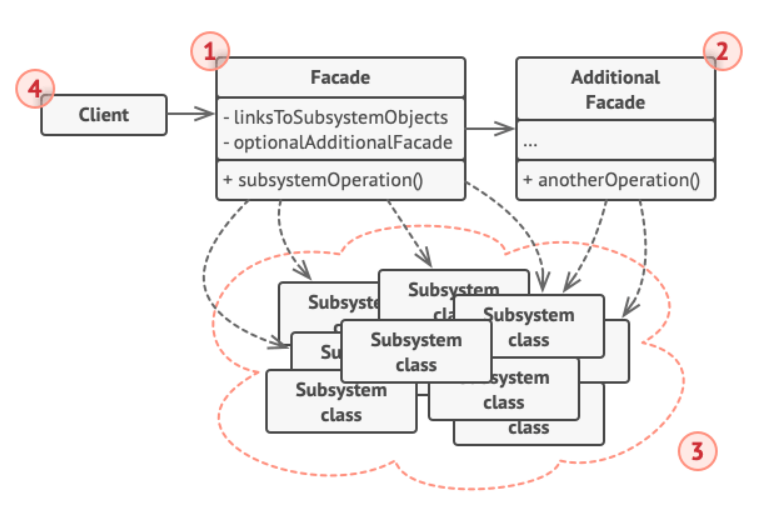
! Це оцінний принцип, який ви застосовуєте при оцінці всіх проектних рішень.



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

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

**Структура:**



1. Фасад надає швидкий доступ до певної функціональності підсистеми. Він «знає», яким класам потрібно переадресувати запит, і які дані для цього потрібні.
2. Додатковий фасад можна ввести, щоб не захаращувати єдиний фасад різнорідною функціональністю. Він може використовуватися як клієнтом, так й іншими фасадами.
3. Складна підсистема має безліч різноманітних класів. Для того, щоб примусити усіх їх щось робити, потрібно знати подробиці влаштування підсистеми, порядок ініціалізації об’єктів та інші деталі.

Класи підсистеми не знають про існування фасаду і працюють один з одним безпосередньо.

1. Клієнт використовує фасад замість безпосередньої роботи з об’єктами складної підсистеми.

**Застосування:**

* Якщо вам потрібно надати простий або урізаний інтерфейс до складної підсистеми.

Часто підсистеми ускладнюються в міру розвитку програми. Застосування більшості патернів призводить до появи менших класів, але у великій кількості. Таку підсистему простіше використовувати повторно, налаштовуючи її кожен раз під конкретні потреби, але, разом з тим, використовувати таку підсистему без налаштовування важче. Фасад пропонує певний вид системи за замовчуванням, який влаштовує більшість клієнтів.

* Якщо ви хочете розкласти підсистему на окремі рівні.

Використовуйте фасади для визначення точок входу на кожен рівень підсистеми. Якщо підсистеми залежать одна від одної, тоді залежність можна спростити, дозволивши підсистемам обмінюватися інформацією тільки через фасади.

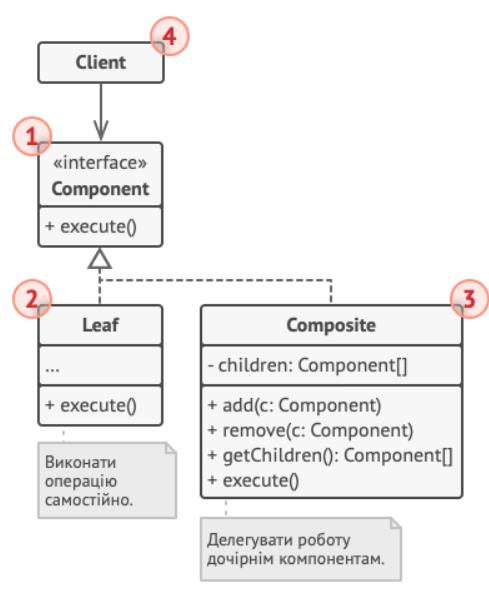
**Основна перевага:** ізолює клієнтів від компонентів складної підсистеми

**Недолік:** фасад ризикує стати божественним об’єктом (антипатерн, який описує об’єкт що зберігає в собі або виконує надто багато операцій), прив’язаним до всіх класів програми.

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

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

**Структура:**



1. Компонент описує загальний інтерфейс для простих і складових компонентів дерева.
2. Лист — це простий компонент дерева, який не має відгалужень. Класи листя міститимуть більшу частину корисного коду, тому що їм нікому передавати його виконання.
3. Контейнер (або *композит*) — це складовий компонент дерева. Він містить набір дочірніх компонентів, але нічого не знає про їхні типи. Це можуть бути як прості компоненти-листя, так і інші компоненти-контейнери. Проте, це не проблема, якщо усі дочірні компоненти дотримуються єдиного інтерфейсу.

Методи контейнера переадресовують основну роботу своїм дочірнім компонентам, хоча можуть додавати щось своє до результату.

1. Клієнт працює з деревом через загальний інтерфейс компонентів.

Завдяки цьому, клієнту не важливо, що перед ним знаходиться — простий чи складовий компонент дерева.

**Застосування:**

* Якщо вам потрібно представити деревоподібну структуру об’єктів.

Патерн Компонувальник пропонує зберігати в складових об’єктах посилання на інші прості або складові об’єкти. Вони, у свою чергу, теж можуть зберігати свої вкладені об’єкти і так далі. У підсумку, ви можете будувати складну деревоподібну структуру даних, використовуючи всього два основних різновида об’єктів.

* Якщо клієнти повинні однаково трактувати прості та складові об’єкти.

Завдяки тому, що прості та складові об’єкти реалізують спільний інтерфейс, клієнту байдуже, з яким саме об’єктом він працюватиме.

**Переваги:**

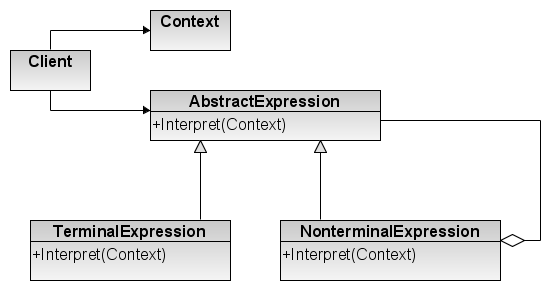
* Спрощує архітектуре клієнта при роботі зі складним деревом компонентів
* Полегшує додавання нових видів компонентів

**Недолік:** створює занадто загальний дизайн класів.

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

**Інтерпретатор** – це поведінковий шаблон проектування, який визначає представлення граматики для заданої мови та інтерпертатор речень цієї мови. Як правило, даний шаблон проектування застосовується для операцій, які часто повторюються. Тобто у разі, якщо якась задача виникає досить часто, є сенс подати її конкретні прояви у вигляді речень простою мовою. Потім можна буде створити інтерпретатор, котрий вирішує задачу, аналізуючи речення цієї мови.

UML діаграма що описує структуру шаблону проєктування Інтерпретатор:



* AbstractExpression — абстрактний вираз:
  + оголошує абстрактну операцію Interpret, загальну для усіх вузлів у абстрактному синтаксичному дереві;
* TerminalExpression — термінальний вираз:
  + реалізує операцію Interpret для термінальних символів граматики;
  + необхідний окремий екземпляр для кожного термінального символу у реченні;
* NonterminalExpression — нетермінальний вираз:
  + по одному такому класу потребується для кожного граматичного правила;
  + зберігає змінні екземпляру типу AbstractExpression для кожного символу;
  + реалізує операцію Interpret для нетермінальних символів граматики. Ця операція рекурсивно викликає себе для змінних, зберігаючих символи;
* Context — контекст:
  + містить інформацію, глобальну по відношенню до інтерпретатору;
* Client — клієнт:
  + будує (або отримує у готовому вигляді) абстрактне синтаксичне дерево, репрезентуюче окреме речення мовою з даною граматикою. Дерево складено з екземплярів класів NonterminalExpression та TerminalExpression;
  + викликає операцію Interpret.

**Застосування**

Шаблон Інтерпретатор слід використовувати, коли є мова для інтерпретації, речення котрої можна подати у вигляді абстрактних синтаксичних дерев. Найкраще шаблон працює коли:

* Граматика проста. Для складних граматик ієрархія класів стає занадто громіздкою та некерованою. У таких випадках краще застосовувати генератори синтаксичних аналізаторів, оскільки вони можуть інтерпретувати вирази, не будуючи абстрактних синтаксичних дерев, що заощаджує пам'ять, а можливо і час;
* Ефективність не є головним критерієм. Найефективніші інтерпретатори зазвичай не працюють безпосередньо із деревами, а спочатку транслюють їх в іншу форму. Так, регулярний вираз часто перетворюють на скінченний автомат. Але навіть у цьому разі сам транслятор можна реалізувати за допомогою шаблону інтерпретатор.

**Переваги:**

* Граматику стає легко розширювати і змінювати, реалізації класів, що описують вузли абстрактного синтаксичного дерева схожі (легко кодуються).
* Можна легко змінювати спосіб обчислення виразів.

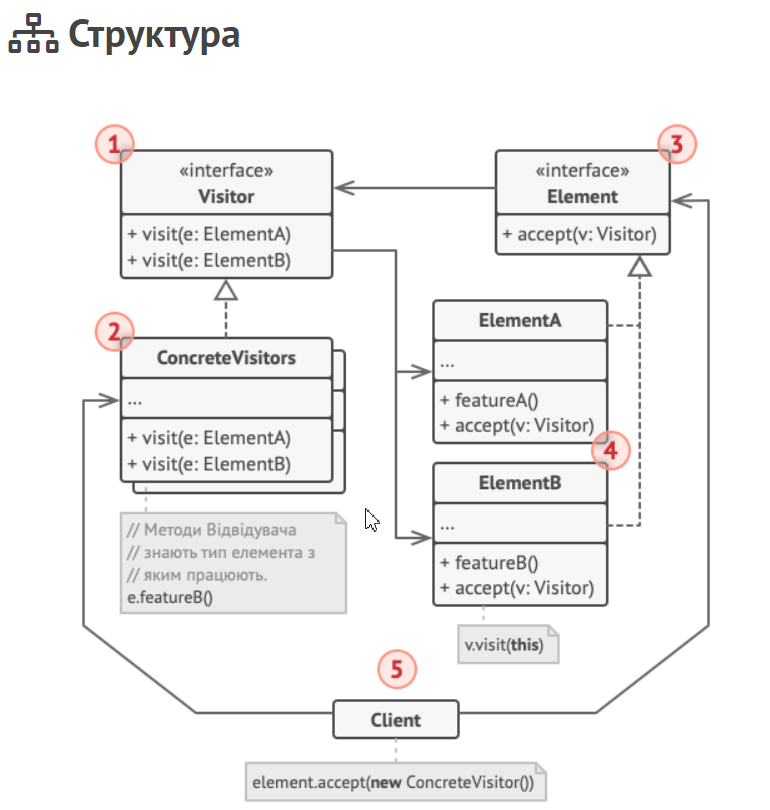
**Недоліки:**

* Супроводжувати граматики з великим числом правил важко.
* Рідко використовується, через свою специфіку

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

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

**Структура:**



1. **Відвідувач** описує спільний для всіх типів відвідувачів інтерфейс. Він оголошує набір методів, що відрізняються типом вхідного параметра. Кожному класу конкретних елементів повинен підходити свій метод. В мовах, які підтримують перевантаження методів, ці методи можуть мати однакові імена, але типи їхніх параметрів повинні відрізнятися.
2. **Конкретні відвідувачі** реалізують якусь особливу поведінку для всіх типів елементів, які можна подати через методи інтерфейсу відвідувача.
3. **Елемент** описує метод прийому відвідувача. Цей метод повинен мати лише один параметр, оголошений з типом загального інтерфейсу відвідувачів.
4. **Конкретні елементи** реалізують методи приймання відвідувача. Мета цього методу — викликати той метод відвідування, який відповідає типу цього елемента. Так відвідувач дізнається, з яким типом елементу він працює.
5. **Клієнтом** зазвичай виступає колекція або складний складовий об’єкт, наприклад, дерево [Компонувальника](https://refactoring.guru/uk/design-patterns/composite). Здебільшого, клієнт не прив’язаний до конкретних класів елементів, працюючи з ними через загальний інтерфейс елементів.

**Застосування**

* Якщо вам потрібно виконати якусь операцію над усіма елементами складної структури об’єктів, наприклад, деревом.

Відвідувач дозволяє застосовувати одну і ту саму операцію до об’єктів різних класів.

* Якщо над об’єктами складної структури об’єктів потрібно виконувати деякі не пов’язані між собою операції, але ви не хочете «засмічувати» класи такими операціями.

Відвідувач дозволяє витягти споріднені операції з класів, що складають структуру об’єктів, помістивши їх до одного класу-відвідувача. Якщо структура об’єктів використовується в декількох програмах, то патерн дозволить кожній програмі мати тільки потрібні в ній операції.

* Якщо нова поведінка має сенс тільки для деяких класів з існуючої ієрархії.

Відвідувач дозволяє визначити поведінку тільки для цих класів, залишивши її порожньою для всіх інших.

**Переваги:**

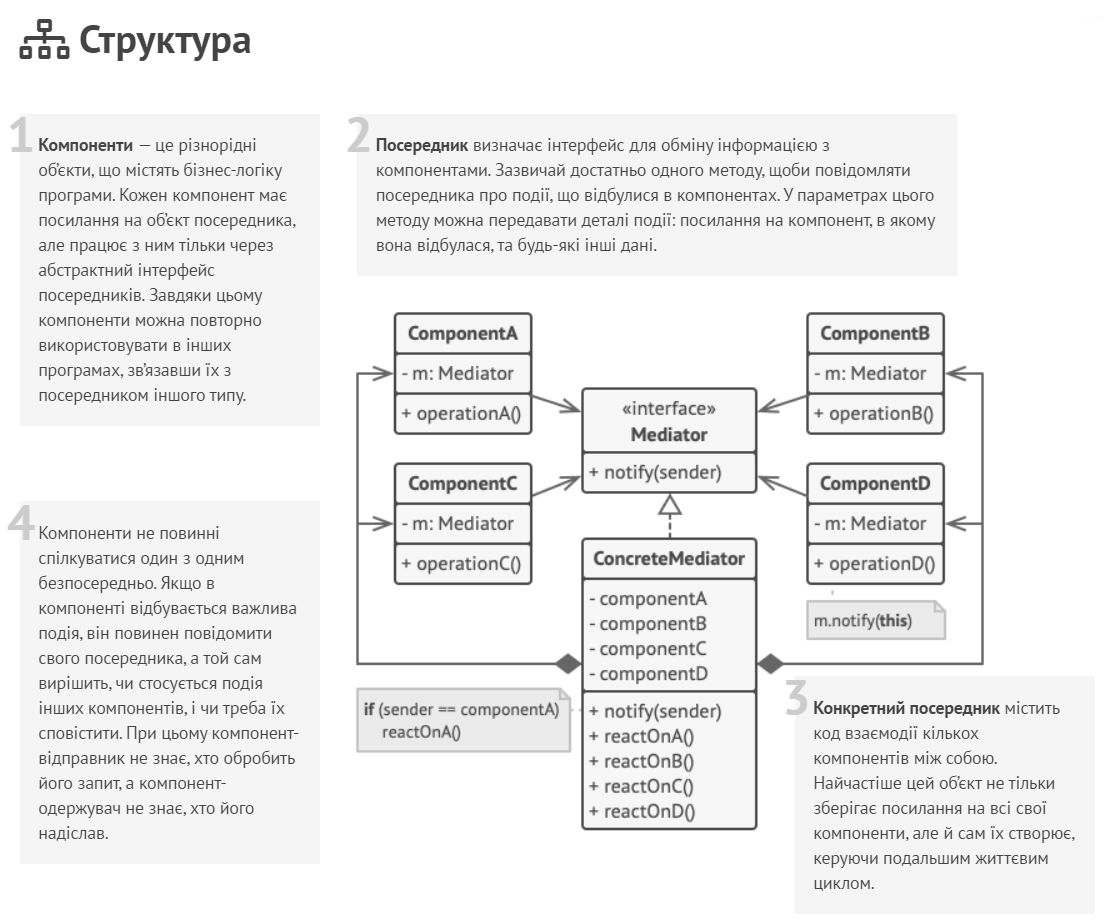
* Спрощує додавання операцій, працюючих зі складними структурами об’єктів.
* Об’єднує споріднені операції в одному класі.
* Відвідувач може накопичувати стан при обході структури елементів.

**Недоліки:**

* Патерн невиправданий, якщо ієрархія елементів часто змінюється.
* Може призвести до порушення інкапсуляції елементів.

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

**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.



**Застосування**

* Коли вам складно змінювати деякі класи через те, що вони мають величезну кількість хаотичних зв’язків з іншими класами.
* Посередник дозволяє розмістити усі ці зв’язки в одному класі. Після цього вам буде легше їх відрефакторити, зробити більш зрозумілими й гнучкими.
* Коли ви не можете повторно використовувати клас, оскільки він залежить від безлічі інших класів.
* Після застосування патерна компоненти втрачають колишні зв’язки з іншими компонентами, а все їхнє спілкування відбувається опосередковано, через об’єкт посередника.
* Коли вам доводиться створювати багато підкласів компонентів, щоб використовувати одні й ті самі компоненти в різних контекстах.
* Якщо раніше зміна відносин в одному компоненті могла призвести до лавини змін в усіх інших компонентах, то тепер вам достатньо створити підклас посередника та змінити в ньому зв’язки між компонентами.

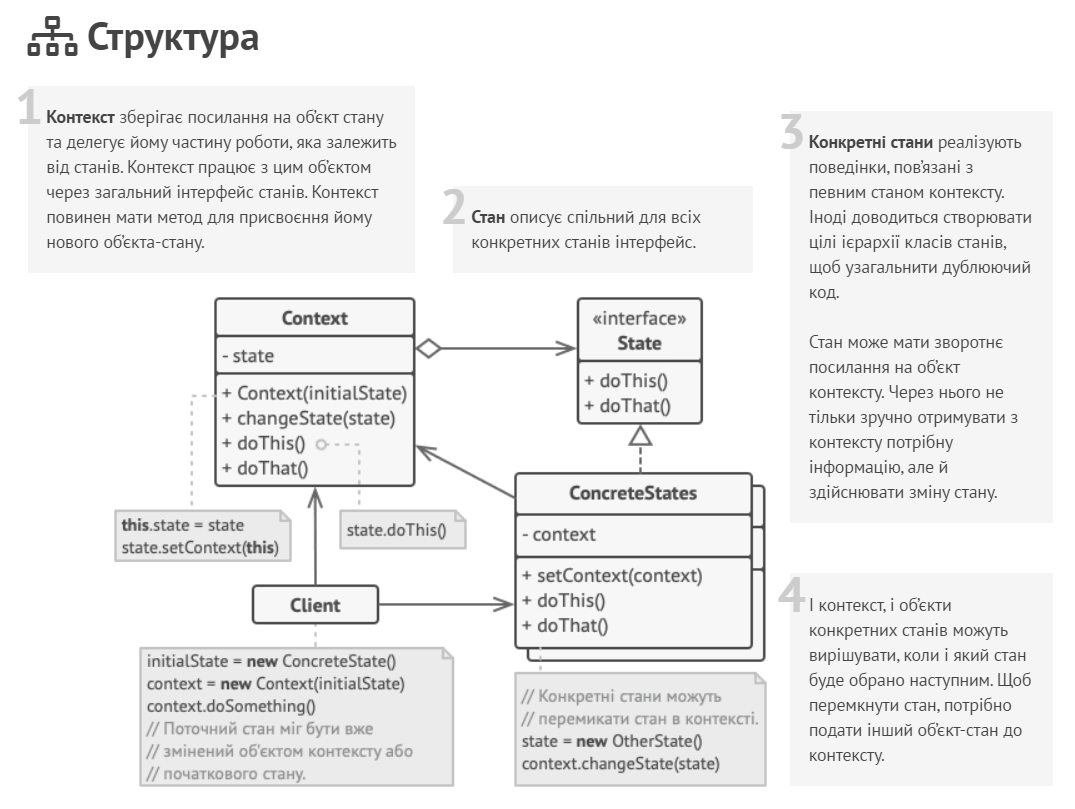
**Переваги:**

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

**Недолік:** посередник може сильно «роздутися».

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

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



**Застосування:**

* Якщо у вас є об’єкт, поведінка якого кардинально змінюється в залежності від внутрішнього стану, причому типів станів багато, а їхній код часто змінюється.

Патерн пропонує виділити в окремі класи всі поля й методи, пов’язані з визначеним станом. Початковий об’єкт буде постійно посилатися на один з об’єктів-станів, делегуючи йому частину своєї роботи. Для зміни стану до контексту достатньо буде підставляти інший об’єкт-стан.

* Якщо код класу містить безліч великих, схожих один на одного умовних операторів, які вибирають поведінки в залежності від поточних значень полів класу

Патерн пропонує перемістити кожну гілку такого умовного оператора до власного класу. Сюди ж можна поселити й усі поля, пов’язані з цим станом.

* Якщо ви свідомо використовуєте табличну машину станів, побудовану на умовних операторах, але змушені миритися з дублюванням коду для схожих станів та переходів.

Патерн Стан дозволяє реалізувати ієрархічну машину станів, що базується на наслідуванні. Ви можете успадкувати схожі стани від одного батьківського класу та винести туди весь дублюючий код.

**Переваги:**

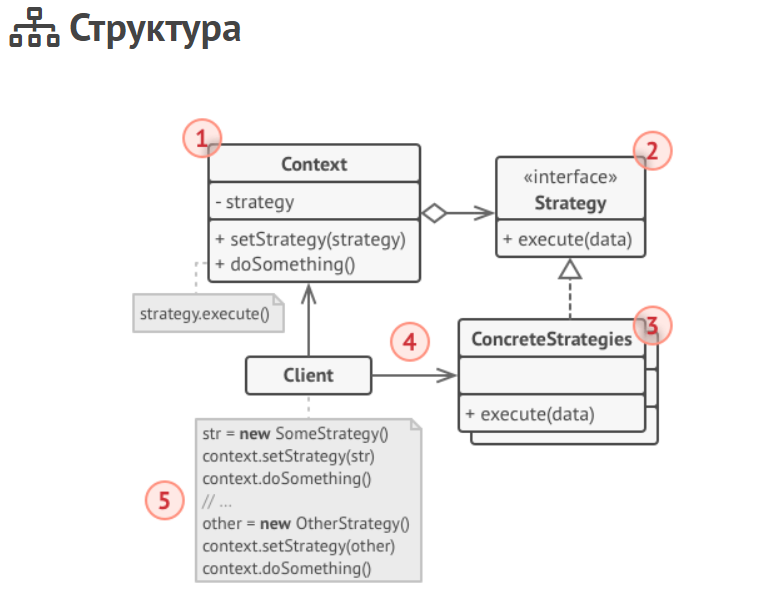
* Позбавляє від безлічі великих умовних операторів машини станів.
* Концентрує в одному місці код, пов’язаний з певним станом.
* Спрощує код контексту.

**Недолік:** може невиправдано ускладнити код, якщо станів мало, і вони рідко змінюються.

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

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

**Структура:**



1. Контекст зберігає посилання на об’єкт конкретної стратегії, працюючи з ним через загальний інтерфейс стратегій.
2. Стратегія визначає інтерфейс, спільний для всіх варіацій алгоритму. Контекст використовує цей інтерфейс для виклику алгоритму.

Для контексту неважливо, яка саме варіація алгоритму буде обрана, оскільки всі вони мають однаковий інтерфейс.

1. Конкретні стратегії реалізують різні варіації алгоритму.
2. Під час виконання програми контекст отримує виклики від клієнта й делегує їх об’єкту конкретної стратегії.
3. Клієнт повинен створити об’єкт конкретної стратегії та передати його до конструктора контексту. Крім того, клієнт повинен мати можливість замінити стратегію на льоту, використовуючи сетер поля стратегії. Завдяки цьому, контекст не знатиме про те, яку саме стратегію зараз обрано.

**Застосування**

* Якщо вам потрібно використовувати різні варіації якого-небудь алгоритму всередині одного об’єкта.

Стратегія дозволяє варіювати поведінку об’єкта під час виконання програми, підставляючи до нього різні об’єкти-поведінки (наприклад, що відрізняються балансом швидкості та споживання ресурсів).

* Якщо у вас є безліч схожих класів, які відрізняються лише деякою поведінкою

Стратегія дозволяє відокремити поведінку, що відрізняється, у власну ієрархію класів, а потім звести початкові класи до одного, налаштовуючи його поведінку стратегіями.

* Якщо ви не хочете оголювати деталі реалізації алгоритмів для інших класів.

Стратегія дозволяє ізолювати код, дані й залежності алгоритмів від інших об’єктів, приховавши ці деталі всередині класів-стратегій.

* Якщо різні варіації алгоритмів реалізовано у вигляді розлогого умовного оператора. Кожна гілка такого оператора є варіацією алгоритму.

Стратегія розміщує кожну лапу такого оператора до окремого класу-стратегії. Потім контекст отримує певний об’єкт-стратегію від клієнта й делегує йому роботу. Якщо раптом знадобиться змінити алгоритм, до контексту можна подати іншу стратегію.

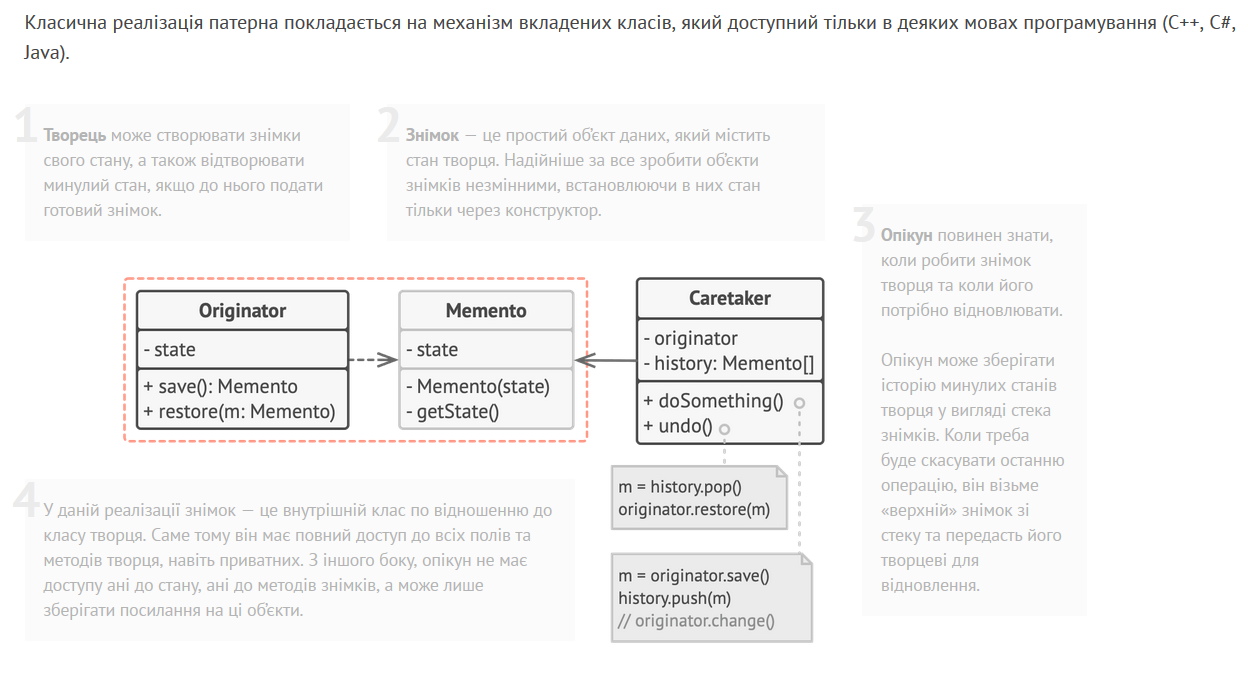
**Переваги:**

* Гаряча заміна алгоритмів на льоту.
* Ізолює код і дані алгоритмів від інших класів.
* Заміна спадкування делегуванням.
* Реалізує принцип відкритості/закритості.

**Недоліки:**

* Ускладнює програму внаслідок додаткових класів.
* Клієнт повинен знати, в чому полягає різниця між стратегіями, щоб вибрати потрібну.

# 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.

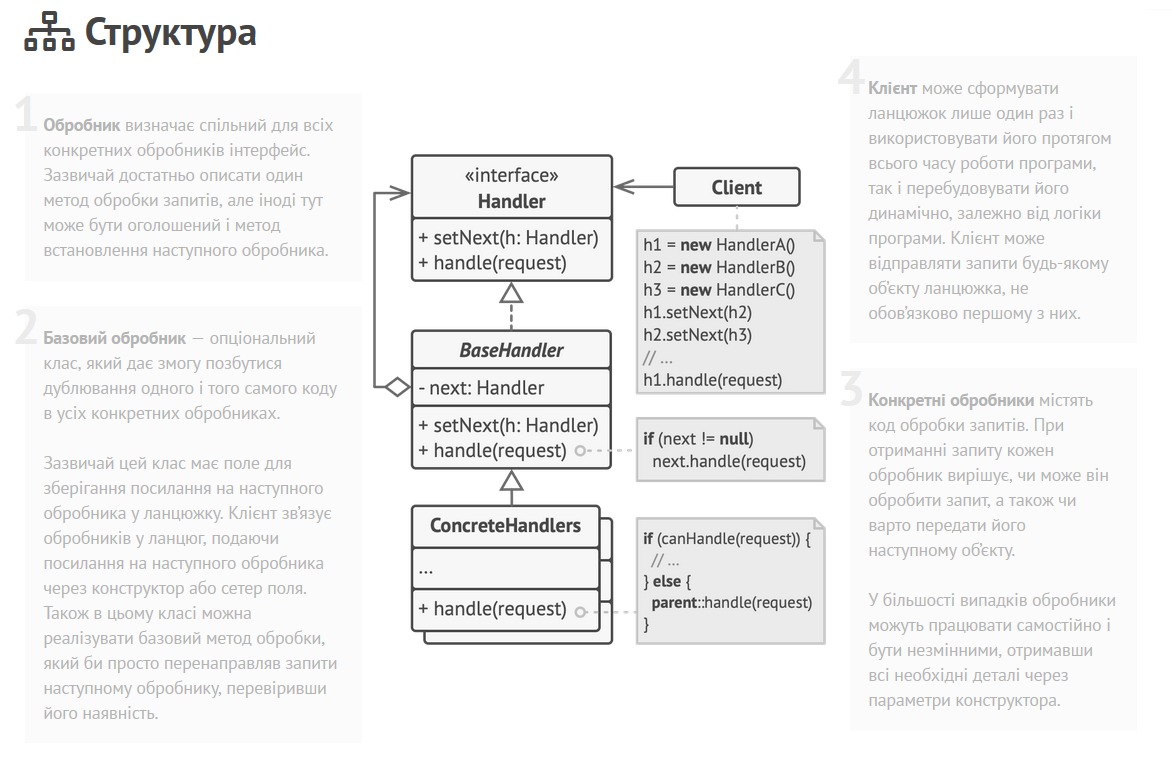
**Переваги:**

* 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.

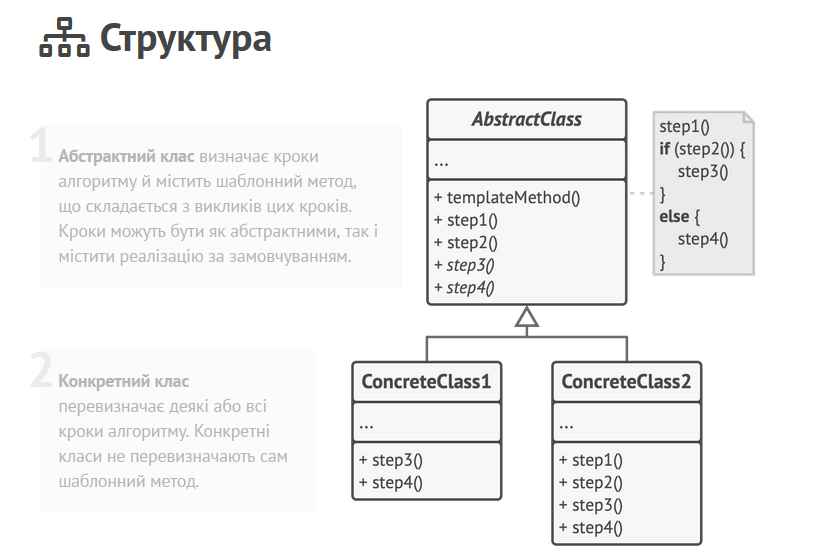
Переваги:

* 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 :

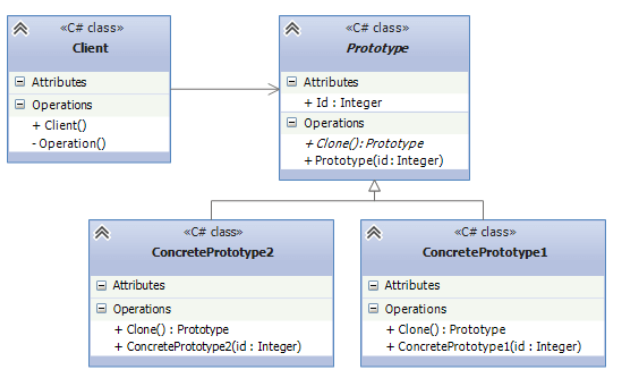
* Полегшує повторне використання коду.

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.

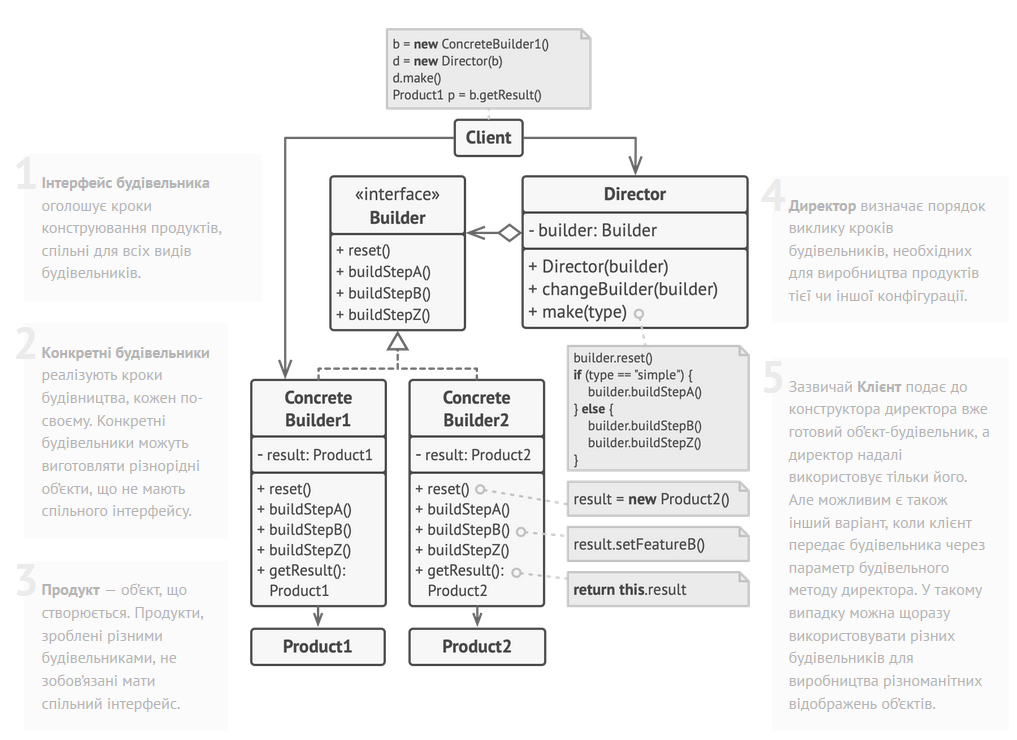
Переваги :

* 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.

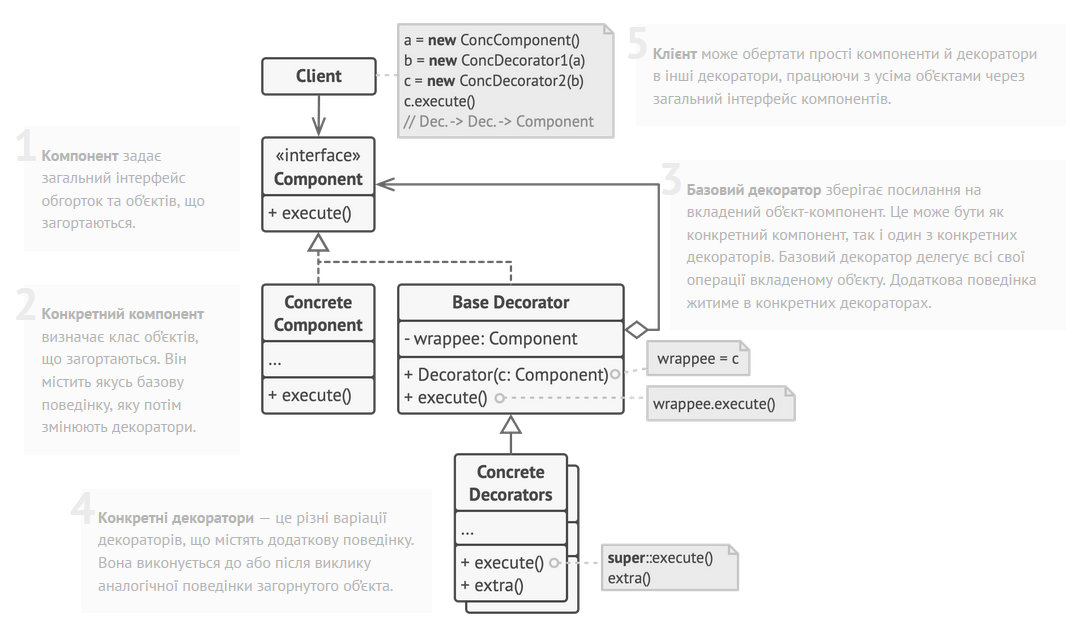
Переваги:

* 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.

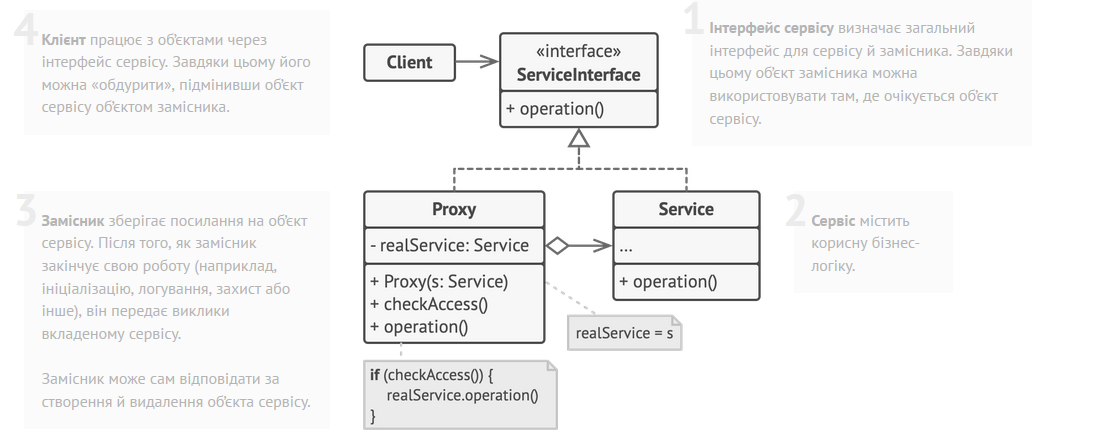
Переваги:

* 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:

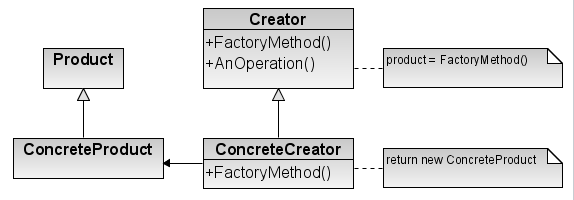
* Дозволяє контролювати сервісний об’єкт непомітно для клієнта.
* Може працювати, навіть якщо сервісний об’єкт ще не створено.
* Може контролювати життєвий цикл службового об’єкта.

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:

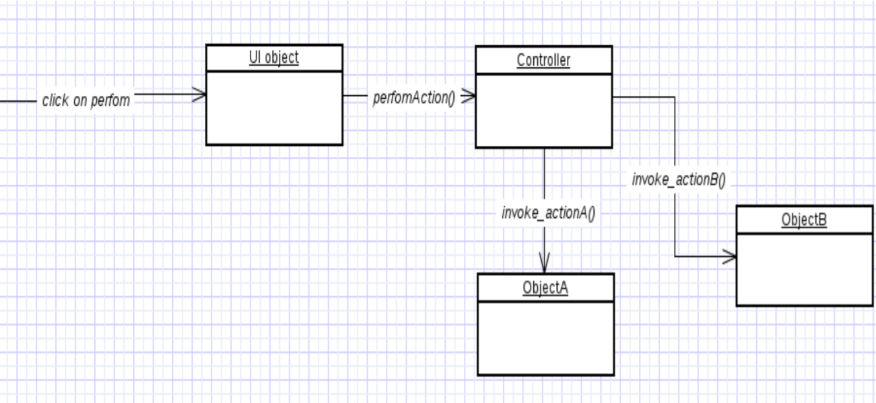
* Позбавляє клас від прив’язки до конкретних класів продуктів.
* Виділяє код виробництва продуктів в одне місце, спрощуючи підтримку коду.
* Спрощує додавання нових продуктів до програми.
* Реалізує принцип відкритості/закритості.

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).

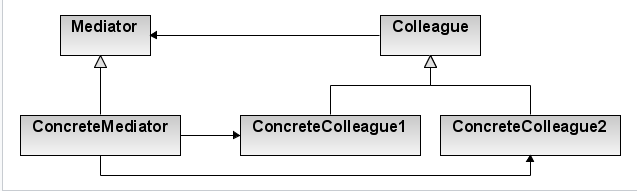
Недоліки:

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

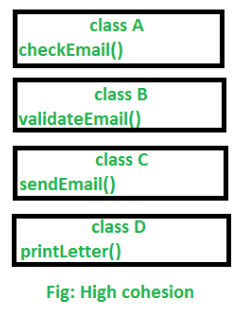
* Усуває залежності між компонентами, дозволяючи використовувати їх повторно.
* Спрощує взаємодію між компонентами.
* Централізує керування в одному місці.

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:

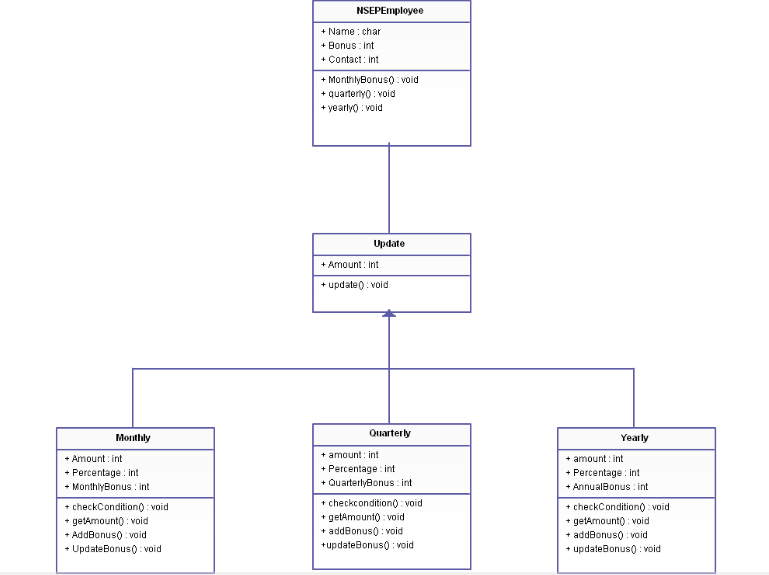
Класи з високим ступенем зачеплення прості в підтримці і повторному використанні.

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.