Лекція 4. Критерії хорошої архітектури. Ієрархія принципів проектування

Слайд 2. **Зміст лекції 4**

4.1 Критерії хорошої архітектури..
4.2 Ієрархія принципів проектування.

4.1 Критерії хорошої архітектури.

Хороша архітектура це перш за все вигідна архітектура, що робить процес розробки і супроводу програми більш простим і ефективним. Програму з хорошою архітектурою легше розширювати і змінювати, а також тестувати, налагоджувати і розуміти. Тобто, насправді можна сформулювати список цілком розумних і універсальних критеріїв:

Ефективність системи. В першу чергу програма, звичайно ж, повинна вирішувати поставлені завдання і добре виконувати свої функції, причому в різних умовах. Сюди можна віднести такі характеристики, як надійність, безпеку, продуктивність, здатність справлятися зі збільшенням навантаження (масштабованість) і т.п.

Гнучкість системи. Будь-який додаток доводиться міняти з часом - змінюються вимоги, додаються нові. Чим швидше і зручніше можна внести зміни в існуючий функціонал, чим менше проблем і помилок це викличе - тим гнучкіше і конкурентоздатною система. Тому в процесі розробки намагайтеся оцінювати те, що виходить, на предмет того, як вам це потім, можливо, доведеться міняти. Запитайте у себе: «А що буде, якщо поточний архітектурне рішення виявиться невірним?», «Яка кількість коду піддасться при цьому змін?». Зміна одного фрагмента системи не повинно впливати на її інші фрагменти. По можливості, архітектурні рішення не повинні «вирубувати в камені», і наслідки архітектурних помилок повинні бути в розумній мірі обмежені. "Гарна архітектура дозволяє ВІДКЛАДАТИ прийняття ключових рішень" (Боб Мартін) і мінімізує «ціну» помилок.

Можливість розширення системи. Можливість додавати в систему нові сутності та функції, не порушуючи її основної структури. На початковому етапі в систему має сенс закладати лише основний і самий необхідний функціонал (принцип YAGNI - you is not gonna need it, «Вам це не знадобиться») Але при цьому архітектура повинна дозволяти легко нарощувати додатковий функціонал в міру необхідності. Причому так, щоб внесення найбільш ймовірних змін вимагало найменших зусилm.

Вимога, щоб архітектура системи володіла гнучкістю і розширюваністю (тобто була здатна до змін і еволюції) є настільки важливим, що воно навіть сформульовано у вигляді окремого принципу - «Принципу відкритості / закритості» (Open-Closed Principle - другий з п'яти принципів SOLID) : Програмні суті (класи, модулі, функції і т.п.) повинні бути відкритими для розширення, але закритими для модифікації. Іншими словами: Повинна бути можливість розширити / змінити поведінку системи без зміни / переписування вже існуючих частин системи.

Це означає, що додаток слід проектувати так, щоб зміна його поведінки і додавання нових функцій був би досягнутий за рахунок написання нового коду (розширення), і при цьому не доводилося б змінювати вже існуючий код. В такому випадку поява нових вимог не спричинить за собою модифікацію існуючої логіки, а зможе бути реалізовано перш за все за рахунок її розширення. Саме цей принцип є основою «плагін архітектури» (Plugin Architecture). Про те, за рахунок яких технік це може бути досягнуто, буде розказано далі.

Масштабованість процесу розробки. Можливість скоротити термін розробки за рахунок додавання до проекту нових людей. Архітектура повинна дозволяти розпаралелити процес розробки, так щоб безліч людей могли працювати над програмою одночасно.

Тестованість. Код, який легше тестувати, буде містити менше помилок і надійніше працювати. Але тести не тільки покращують якість коду. Багато розробників приходять до висновку, що вимога «хорошою тестованості» є також спрямовуючою силою, автоматично веде до хорошого дизайну, і одночасно одним з найважливіших критеріїв, що дозволяють оцінити його якість: «Використовуйте принцип« тестованості »класу в якості« лакмусового папірця »хорошого дизайну класу. Навіть якщо ви не напишіть жодного рядка тестового коду, відповідь на це питання в 90% випадків допоможе зрозуміти, наскільки все «добре» або «погано» з його дизайном»

Існує ціла методологія розробки програм на основі тестів, яка так і називається - Розробка через тестування (Test-Driven Development, TDD).

Можливість повторного використання. Систему бажано проектувати так, щоб її фрагменти можна було повторно використовувати в інших системах.

Добре структурований, читаємий і зрозумілий код.

Супровід. Над програмою, як правило, працює безліч людей - одні йдуть, приходять нові. Після написання супроводжувати програму теж, як правило, доводиться людям, які не брав участь в її розробці. Тому хороша архітектура повинна давати можливість відносно легко і швидко розібратися в системі нових людей. Проект повинен бути добре структурований, не містити

дублювання, мати добре оформлений код і бажано документацію. І по можливості в системі краще застосовувати стандартні, загальноприйняті рішення звичні для програмістів. Чим екзотичніша система, тим складніше її зрозуміти іншим (Принцип найменшого подиву - Principle of least astonishment. Зазвичай, він використовується відносно призначеного для користувача інтерфейсу, але застосуємо і до написання коду).

Ну і для повноти критерії поганого дизайну:

1. Його важко змінити, оскільки будь-яка зміна впливає на занадто велику кількість інших частин системи. (Жорсткість, Rigidity).

2. При внесенні змін несподівано ламаються інші частини системи. (Крихкість, Fragility).

3. Код важко використовувати повторно в іншому додатку, оскільки його занадто важко «виплутати» з поточної програми. (Нерухомість, Immobility).

4.2 Ієрархія Принципів проектування

Базове правило для розташування принципів проектування в ієрархію буде наступним: якщо ви можете дотримуватися принципу А, але відмовилися або не знали про прийом В, то А є найбільш фундаментальним принципом проектування. *Наприклад*, ви можете розібратися з ООП, але поки не прочитати про шаблони проектування Gang of Four-це не завадить вам писати програми з використанням ООП; тому ООП є найбільш фундаментальним принципом. Або, наприклад, ви можете відмовитися від ООП (або від якихось його атрибутів - наприклад, успадкування та поліморфізму), але, тим не менш, продовжуєте писати модульні програми на С з чіткими інтерфейсами модулів і рівнями абстракції. Тому модульність і абстракція є більш базовими принципами, ніж ООП.

Важность управления сложностью

Программные проекты редко терпят крах по техническим причинам. Чаще всего провал объясняется неадекватной выработкой требований, неудачным планированием или неэффективным управлением. Если же провал обусловлен все-таки преимущественно технической причиной, очень часто ею оказывается неконтролируемая сложность. Иначе говоря, приложение стало таким сложным, что разработчики перестали по-настоящему понимать, что же оно делает. Если работа над проектом достигает момента, после которого уже никто не может полностью понять, как изменение одного фрагмента программы повлияет на другие фрагменты, прогресс прекращается.

> Есть два способа разработки проекта приложения: сделать его настолько простым, чтобы было *очевидно*, что в нем нет недостатков, или сделать его таким сложным, чтобы в нем не было *очевидных* недостатков.
>
> *Ч. Э. Р. Хоар (C. A. R. Hoare)*

Управление сложностью — самый важный технический аспект разработки ПО. По-моему, *управление сложностью* настолько важно, что оно должно быть Главным Техническим Императивом Разработки ПО.

KISS (акронім для «**Keep it short and simple**») — принцип проектування, прийнятий в ВМС США в 1960. Принцип KISS стверджує, що більшість систем працюють найкраще, якщо вони залишаються простими, а не ускладнюються. **Тому в області проектування простота повинна бути однією з ключових цілей і слід уникати непотрібної складності.**

KISS - принцип, що забороняє використання більш складних засобів, ніж необхідно.

• Розбивайте завдання на підзадачі які не повинні на вашу думку тривати більше 4-12 годин написання коду

• Розбивайте завдання на безліч дрібніших завдань, кожне завдання має вирішуватися одним або парою класів

• Зберігайте ваші методи маленькими. Кожен метод має складатися не більше ніж з 30-40 рядків. Кожен метод має вирішувати одну маленьку задачу, а не безліч випадків. Якщо у вашому методі безліч умов, розбийте його на

кілька. Це підвищить читаність, дозволить легше підтримувати код і швидше знаходити помилки в ньому. Ви полюбите покращувати код.

• Зберігайте ваші класи маленькими. Тут застосовується та ж техніка що і з методами.

• Придумайте рішення задачі спочатку, потім напишіть код. Ніколи не робіть інакше. Багато розробники придумують рішення задачі під час написання коду і в цьому немає нічого поганого. Ви можете робити так і при цьому дотримуватися вище зазначеного правила. Якщо ви можете в розумі розбивати задачу на більш дрібні частини, коли ви пишете код, робіть це будь-якими способами. І не бійтеся переписувати код ще і ще і ще ... В рахунок не йде число рядків, до тих пір поки ви вважаєте що можна ще менше / ще краще.

• Не бійтеся позбавлятися від коду. Зміна старого коду і написання нового рішення два дуже важливих моменти. Якщо ви зіткнулися з новими вимогами, або були сповіщені про них раніше, тоді деколи краще придумати нове більш витончене рішення вирішальне і старі і нові завдання.

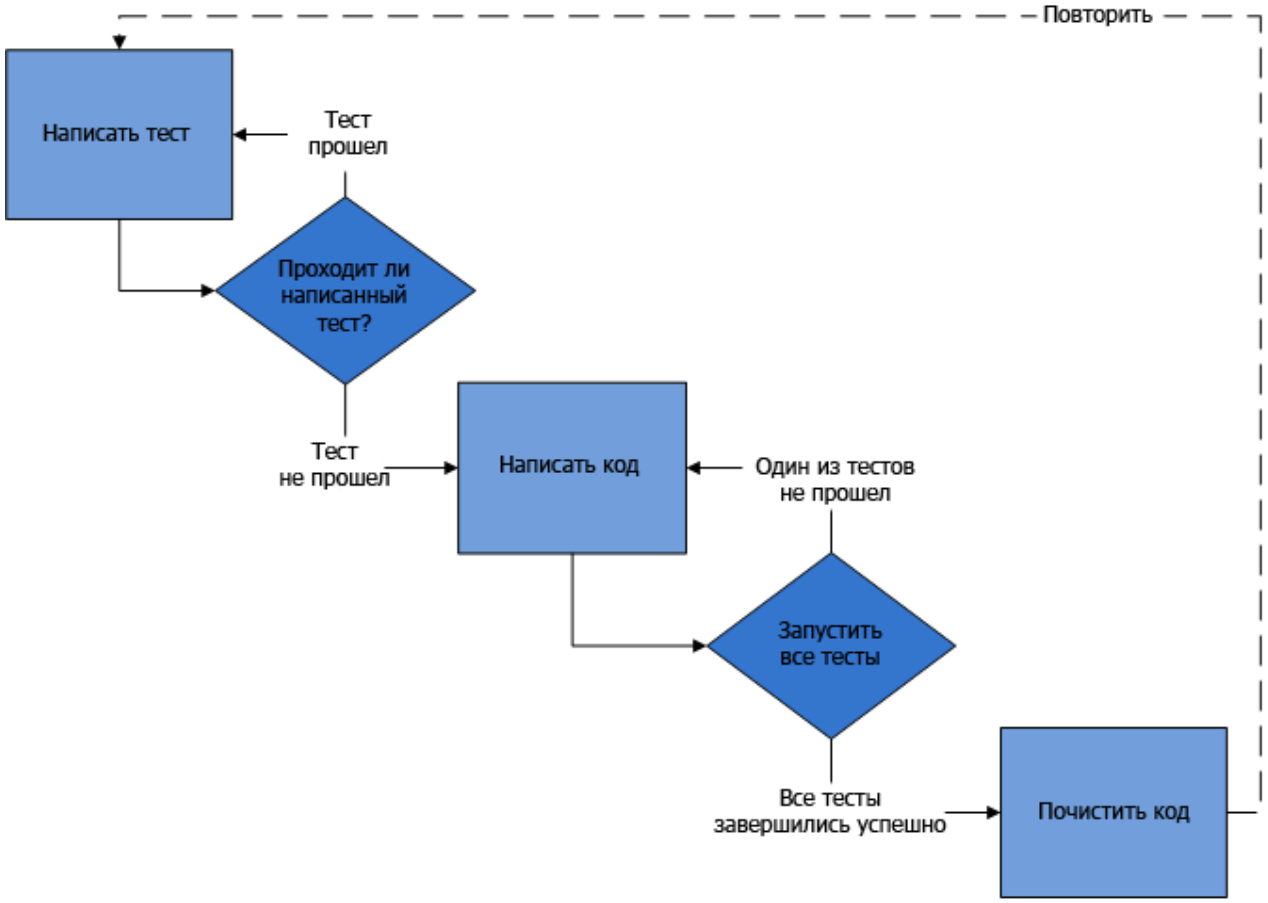
Do not repeat yourself, DRY (укр. Не повторювати) - це принцип розробки програмного забезпечення, націлений на зниження повторення інформації різного роду, особливо в системах з великою кількістю шарів абстрагування. Принцип DRY формулюється як: «Кожна частина знання повинна мати єдине, несуперечливе і авторитетне уявлення в рамках системи» [1]. Він був сформульований Енді Хантом (англ.) І Дейвом Томасом (англ.) В їх книзі The Pragmatic Programmer (англ.). Вони застосовували цей принцип до «схемами баз даних, планам тестування, збірок програмного забезпечення, навіть до документації». Коли принцип DRY застосовується успішно, зміна єдиний елемент системи не вимагає внесення змін в інші, логічно не пов'язані елементи. Ті елементи, які логічно пов'язані, змінюються передбачувано і одноманітно.

Предметно-орієнтоване проектування (рідше проблемно-орієнтоване, англ. Domain-driven design, DDD) - це набір принципів і схем, спрямованих на створення оптимальних систем об'єктів. Зводиться до створення програмних абстракцій, які називаються моделями предметних областей. У ці моделі входить бізнес-логіка, що встановлює зв'язок між реальними умовами області застосування продукту і кодом.

Предметно-орієнтоване проектування не є який-небудь конкретної технологією або методологією. DDD - це набір правил, які дозволяють приймати правильні проектні рішення. Даний підхід дозволяє значно прискорити процес проектування програмного забезпечення в незнайомій предметної області.

Підхід DDD особливо корисний в ситуаціях, коли розробник не є фахівцем в області розробляється продукту. Наприклад: програміст не може знати все області, в яких потрібно створити ПО, але за допомогою правильного уявлення структури, за допомогою проблемно-орієнтованого підходу, може без праці спроектувати додаток, грунтуючись на ключових моментах і знаннях робочої області.

Розробка через тестування (test-driven development, TDD) - техніка розробки програмного забезпечення, яка ґрунтується на повторенні дуже коротких циклів розробки: спочатку пишеться тест, що покриває бажану зміну, потім пишеться код, який дозволить пройти тест, і під кінець проводиться рефакторинг нового коду до відповідних стандартів. Кент Бек, який вважається винахідником цієї техніки, стверджував в 2003 році, що розробка через тестування заохочує простий дизайн і вселяє впевненість.



Додавання теста

При розробці через тестування, додавання кожної нової функціональності (англ. Feature) в програму починається з написання тесту. Неминуче цей тест не буде проходити, оскільки відповідний код ще не написаний. (Якщо ж написаний тест пройшов, це означає, що або запропонована «нова» функціональність вже існує, або тест має недоліки.) Щоб написати тест,

розробник повинен чітко розуміти пред'являються до нової можливості вимоги. Для цього розглядаються можливі сценарії використання і призначені для користувача історії. Нові вимоги можуть також спричинити зміну існуючих тестів. Це відрізняє розробку через тестування від технік, коли тести пишуться після того, як код вже написаний: вона змушує розробника сфокусуватися на вимогах до написання коду - тонке, але важлива відмінність.

Запуск всіх тестів: переконатися, що нові тести не проходять

На цьому етапі перевіряють, що щойно написані тести не проходять. Цей етап також перевіряє самі тести: написаний тест може проходити завжди і відповідно бути марним. Нові тести повинні не проходити по зрозумілих причин. Це збільшить впевненість (хоча не гарантуватиме повністю), що тест справді тестує то, для чого він був розроблений.

Написати код

На цьому етапі пишеться новий код так, що тест буде проходити. Цей код не обов'язково повинен бути ідеальний. Припустимо, щоб він проходив тест якимось неелегантно способом. Це прийнятно, оскільки наступні етапи поліпшать і відполірують його.

Важливо писати код, призначений саме для проходження тесту. Не слід додавати зайвої і, відповідно, не тестується функціональності.

Запуск всіх тестів: переконатися, що всі тести проходять

Якщо всі тести проходять, програміст може бути впевнений, що код задовольняє всім тестованим вимогам. Після цього можна приступити до завершального етапу цикли.

Рефакторінг

Коли досягнута необхідна функціональність, на цьому етапі Код може бути почищений. Рефакторинг - процес зміни внутрішньої структури програми, що не зачіпає її зовнішньої поведінки і має на меті полегшити розуміння її роботи, усунути дублювання коду, полегшити внесення змін в найближчому майбутньому.

Повторити цикл

Описаний цикл повторюється, реалізуючи все нову і нову функціональність. Кроки слід робити невеликими, від 1 до 10 змін між запусками тестів. Якщо новий код не задовольняє новими тестами або старі тести перестають проходити, програміст повинен повернутися до налагодження. При використанні сторонніх бібліотек не слід робити настільки невеликі зміни, які буквально тестують саму сторонню бібліотеку, а не код, її

використовує, якщо тільки немає підозр, що бібліотека містить помилки.

Запитання до Лекції 4.

1. Критерії хорошої архітектури.
2. Сформулюйте список цілком розумних і універсальних критеріїв архітектури.
3. Ієрархія принципів проектування
4. Охарактеризуйте ефективність та гнучкість системи.
5. Що розуміється під можливістю розширення системи
6. Охарактеризуйте масштабованість процесу розробки.
7. Що розуміється під тестуванням.
8. Охарактеризуйте можливість повторного використання.
9. Що розуміється під супроводом.
10. Охарактеризуйте критерії поганого дизайну.
11. Зобразіть графічно та опишіть ієрархію принципів проектування
12. Що розуміється під KISS (акронім для «Keep it short and simple») — принципом проектування, прийнятим в ВМС США в 1960?
13. Охарактеризуйте Do not repeat yourself, DRY (*укр. Не повторювати*)
14. Охарактеризуйте предметно-орієнтоване проектування
15. Охарактеризуйте предметно-орієнтоване проектування
16. Охарактеризуйте підхід DDD
17. Розробка через тестування (test-driven development, TDD)
18. Охарактеризуйте додавання теста

19.Запуск всіх тестів: переконатися, що нові тести не проходять
20.Охарактеризуйте написання коду
21.Запуск всіх тестів: переконатися, що всі тести проходять
22.Охарактеризуйте рефакторінг
23.Що розуміється під Повторенням циклу?