|  |
| --- |
| Міністерство освіти та науки Українинаціональний університет «Одеська політехніка»інститут штучного інтелекту та робототехніки**Кафедра програмних та комп'ютерно-інтегрованих технологій**Методичні вказівки з дисциплініСучасна технологія розробки програмного забезпечення(Теоретична частина)Для студентів інституту штучного інтелекту та робототехнікиІнший (магістерський) рівень вищої освітиСпеціальність: 151 – Автоматизація та комп'ютерно-інтегровані технологіїОсвітньо-професійна програма: Комп'ютерні технології автоматизації;Кількість рік/кредитів ЄКТС за навчальним планом: 135/4,5 |
|  | Схвалено на засіданні кафедри ПКІТ протокол №7 від 26.01.2022р. |
| **Одеса 2022** |

Методичні вказівки з дисципліни «Сучасна технологія розробки програмного забезпечення» (Теоретична частина) для студ. спец. 151 – «Автоматизація та комп'ютерно-інтегровані технології» ден. та заоч. форм навч./уклад.: В.О. Давидов - ОП, 2022. - 54 с.

|  |  |
| --- | --- |
| Укладачі: | **В.О. Давидів**, Канд. техн. наук |
|  |  |
|  |  |
|  |  |
|  |  |

Зміст

[1 Процес 4](#_Toc97547051)

[1.1. Спіральна модель процесу. 4](#_Toc97547052)

[1.2. Інкрементальна модель процесу. 5](#_Toc97547053)

[2 управління проектом 10](#_Toc97547054)

[3 Аналіз вимог 14](#_Toc97547055)

[4 Аналіз вимог. Завершення SRS: додавання детальних вимог. 16](#_Toc97547056)

[6 Детальне проектування 20](#_Toc97547057)

[6.1. Введення у детальне проектування 20](#_Toc97547058)

[6.2. Діаграми послідовності та діаграми потоків даних у детальному проектуванні 23](#_Toc97547059)

[6.3. Специфікація класів та функцій 23](#_Toc97547060)

[6.4. Специфікація алгоритмів 24](#_Toc97547061)

[6.6. Бібліотека стандартних шаблонів (STL) C++ 26](#_Toc97547062)

[6.8. Вплив детального проектування на проект 27](#_Toc97547063)

[6.9. Детальне проектування та якість 28](#_Toc97547064)

[7 Реалізація модулів 30](#_Toc97547065)

[7.1 Введення у реалізацію 30](#_Toc97547066)

[7.2 Визначення та мета реалізації 30](#_Toc97547067)

[7.3 Типова схема процесу реалізації модулів 31](#_Toc97547068)

[7.4 Реалізація в USDP 32](#_Toc97547069)

[7.5 Мови програмування 34](#_Toc97547070)

[7.6 Програмування та стиль 35](#_Toc97547071)

[7.7 Стандарти програмування 38](#_Toc97547072)

[8 Модульне тестування 45](#_Toc97547073)

[8.1. Введення у модульне тестування 45](#_Toc97547074)

# 1 Процес

Система розробки програмного забезпечення включає персонал, процес, проект і продукт. Всі вони важливі, жоден з них не є важливішим за інших. Цялина будь-якого програмного проекту полягає у виробництві деякого програмного продукту (наприклад, текстового редактора). Те, як у рамках проекту виробляється продукт, є процесом. Оскільки критичним успіху справи є взаємодія членів команди.



Класичною моделлю процесу розробки програм є водоспадна модель, у межах якої процес є послідовністю фаз аналізу вимог, проектування, реалізації, інтеграції та тестування.

Аналіз вимог полягає у зборі вимог до товару. Результатом аналізу, зазвичай, є деякий текст.

Проектування визначає внутрішню структуру товару. Зазвичай такий опис дається у формі діаграм та текстів

Реалізація – це програмування. Результатом реалізації є програмний код усіх рівнів, будь то код, що генерується високорівневою системою програмування, компілятором мови четвертого покоління або будь-якою іншою.

Інтеграція - це процес складання всього товару з окремих елементів.

Насправді перелічені фази не випливають суворо послідовно один за одним, а частково перекриваються. На практиці будь-яку з фаз можна розпочинати до того, як буде повністю завершено попередню.

Водоспадний процес починається з визначення вимог, що висуваються додатком, далі переходить у фазу проектування, потім реалізації і, нарешті, тестування. Іноді водопадний процес включається фаза супроводу. Розробка програмного забезпечення рідко протікає строго у «водопадній» послідовності. У чистому вигляді водопадний процес застосовується досить рідко, хіба що у разі невеликих проектів або коли команда реалізує проект, дуже схожий на один із тих, що були здійснені нею раніше. Основною причиною незастосовності водоспадного процесу у чистому вигляді є складність більшості додатків. Водоспадний процес є основою більшості інших різновидів процесу.

Процеси, у яких водоспадна схема застосовується багаторазово, називаються ітеративними. Відразу обмовимося, що в ітеративних процесах не обов'язково всі кроки схеми водоспаду повинні виконуватися на кожній ітерації.

## 1.1. Спіральна модель процесу.

У разі спірального процесу послідовність аналіз вимог - проектування - реалізація - тестування виконується більше одного разу. І тому може бути кілька причин. Основна причина зазвичай пов'язана з необхідністю попередження ризиків. Іншою причиною може бути необхідність надати замовнику часткову версію проекту для отримання відгуків та побажань. Якщо програма, що розробляється, досить складна, необхідно виконати проміжні інтеграції, не відкладаючи цю фазу на самий кінець, як це пропонує водоспадна модель. Загальна ж ідея спірального процесу полягає в тому, щоб на кожній ітерації будувати чергову версію програми, використовуючи як основу її попередню версію. У цьому випадку процес набуває спіралеподібного характеру



Додаткова перевага ітеративних процесів полягає у можливості збирати на кожній ітерації метричні характеристики процесу. Наприклад, маючи дані про час, який потрібно для виконання першої ітерації, ми можемо уточнити план трафік подальшої роботи. Така можливість є особливо корисною для організацій, які мають невеликий досвід планування розробок.

Хоча спіральна модель відображає типову схему процесу розробки, вона вимагає більш майстерного керування, ніж проста водоспадна модель. Одна з труднощів полягає у підтримці цілісності документації, яка має бути повністю оновлена ​​та доповнена до кінця кожної ітерації. Зокрема, кожна версія програмного коду повинна реалізовувати документований проект і задовольняти документовані вимоги. І все ж таки для більшості програмних проектів переваги спірального процесу переважують його недоліки. Тут можна послатися на досвід Міністерства оборони США, яке, визнавши ці переваги, у 80-х роках відмовилося від прийнятої раніше установки на використання простої водоспадної моделі у всіх програмних проектах.

Скільки ж ітерацій потрібно у разі застосування спіральної моделі? Це залежить від ситуації. Скажімо, типовий проект, трудомісткість якого оцінюється в три людино-місяці, а тривалість — у чотири місяці, найімовірніше. вимагатиме дві-три ітерації. Витрати для проведення більшої кількості кроків можуть легко переважити вигоду від додаткових ітерацій. Випадок, коли кількість ітерацій зростає настільки, що кожна нова ітерація надає надто малу кількість нових можливостей порівняно з попередньою, ми називатимемо інкрементальною розробкою.

## 1.2. Інкрементальна модель процесу.

Іноді можна потроху просувати проект вперед при практично безперервному процесі. Така модель процесу особливо корисна на пізніх стадіях проекту, коли продукт знаходиться на супроводі або коли продукт, що розробляється, дуже схожий зі створеним раніше. Для підтримки відповідного рівня інкрементальної розробки необхідно мати чітко встановлену архітектуру проекту та виключно синхронізовану систему документації. Для організації інкрементальної розробки зазвичай вибирається характерний часовий інтервал, наприклад, тиждень. Потім протягом цього інтервалу відбувається оновлення вихідного проекту (документації, набору тестів, програмного коду тощо). Теоретично кроки розробки можуть виконуватись і паралельно, але такий процес дуже складно скоординувати. Інкрементальна технологія проходить найкраще,



Верифікація та валідація є складовою плану контролю якості. Верифікація відповідає питанням «Чи правильно побудований наш об'єкт?». Або більш детально: «Чи ми на даній фазі в точності те, що було заплановано в попередній фазі. Валідація ж відповідає на запитання: «Чи робимо ми те, що потрібно?». Або іншими словами: ‹ Чи відповідає побудований об'єкт побажанням та потребам замовника?›. Різницю між цими поняттями ілюструє рис. 1.18.



Наведемо найпростіший приклад. Замовник хоче, щоб ми розробили додаток, який вирішує лінійні рівняння виду ах + 12 = с. Ми формулюємо це побажання як наступні детальні вимоги.

Користувач вводить числа а, 1) та с. що складаються не більше ніж із десяти цифр, включаючи максимум чотири цифри після коми.

Додаток шукає рішення уршзнения ах + 11 = з точністю до 1/1000г

Потім ми пишемо програму. реалізую ці вимоги.

Валідація нашої програми полягає у прогоні набору тестів. Наприклад, ми вводимо а = 1, 12 = 1, з – 1 і перевіряємо. що програма видає х

Тестування виявляє наявність будь-яких дефектів у програмі, але не дозволяє переконатися, що дефектів у програмі відсутні. Іншими словами, ми не можемо гарантувати, що наша програма не містить дефектів.

Верифікація полягає в тому, що ми аналізуємо процес побудови програми та переконуємось. що в цьому процесі все зроблено правильно. Процес розробки нашої програми включає наступні кроки.

Отримання вимог замовника.

Підготовка детальних вимог

Підготовка програмного коду.

Тестування програми.

Верифікація полягатиме у наступних перевірках.

Чи відповідають детальні вимоги до цього. що замовник справді хоче? Тут верифікація може включати такі пункти: чи може бути а = 0? Якщо так, то що в цьому випадку має робити програма? Якщо ні, то можливо замовнику потрібно вирішувати рівняння виду х + 12 - 6; взагалі, можливо. що замовнику природніше задавати лінійне рівняння у якомусь іншому вигляді; обговорення із замовником питання точності завдання вхідних даних та одержуваного рішення та т.д.

Інспектування вимог відноситься до процесу верифікації.

Чи реалізує код програми сформульовані вимоги? Це

Має на увазі інспектованість коду, в процесі якого код аналізується з погляду відповідності кожній окремій вимогі. Аналіз може включати математичні докази або грунтуватися на них.

Серед переваг те. що процедури контролю якості можуть бути специфіковані одночасно для всієї організації, гарантуючи цим уніфікованість товарів організації. Це дозволило б заощадити час на переробці та адаптації процедур із проекту до проекту.

Недоліком є ​​те, що неможливо передбачити всі деталі плану до моменту затвердження плану проекту. (Наприклад, ми не можемо скласти розклад верифікації та валідації.)

Стандарти документації заощаджують час на обдумування структури документа. Вони нагадують про важливі речі, які можна проґавити. Використання стандартів уніфікує документацію в організації. Недолік використання стандартів у тому, що розробники іноді почуваються зобов'язаними написати «хоч що-небудь» у стандартний розділ документа, тоді як це можливо. що цей розділ стандарту неадекватний даному конкретному випадку. В результаті виникає відчуття безплідної «паперової» роботи. З іншого боку, розробники дещо скуті стандартом і соромляться додати нестандартний розділ у документ, хоча може бути, він був би дуже важливим у конкретному випадку (наприклад, стандарти 1ЕЕЕ не містять деяких розділів, дуже корисних при розробці вей-додатків).

Метрики – це кількісні значення показників процесу розробки. Вони важливі, оскільки вносять точність у процес розробки, перетворюючи його з шаманства на інженерну діяльність.

Використання метриків. тобто кількісні характеристики, характерні для різних інженерних дисциплін. Наприклад, транспортні потоки характеризуються кількістю автомобілів на годину, а в механіці говорять про граничні навантаження. У проектуванні програм також використовують різні кількісні характеристики, такі як число рядків коду, число класів, кількість дефектів, виявлених за місяць, число функцій у класі.

Трактування метрик визначається контекстом. у якому ці метрики використовуються. Наприклад, якщо ми маємо дві програми, які мають однакову функціональність, однаково ефективні. однаково надійні і добре структуровані, то кращою, мабуть, слід вважати ту з них, яка має менше рядків коду. В іншому ж контексті більше рядків коду може означати більшу продуктивність. За наявності достатньої статистичної бази метрика число рядків коду може бути дуже корисним, особливо у поєднанні з іншими характеристиками. Наприклад, якщо вважати, що метрика надійності програми залишається на потрібному рівні, а збільшення кількості рядків коду відображає розширення можливостей програми, стає зрозумілим, що чим більше рядків коду буде написано за годину, тим краще.

Крім того, слід застосовувати відносну оцінку якості роботи за шкалою від 0 до 10.

Попередні або бажані значення метрик прогнозуються заздалегідь, а потім порівнюються з отриманими результатами.

Індивідуальний процес розробки програмного забезпечення (РЗР). Хемфрі дуже майстерно визначив, які саме навички повинен мати компетентний інженер — розробник програмного забезпечення. Вони описані в індивідуальному процесі розробки програмного забезпечення, який «надає детальні описи методів планування та оцінки, показує розробникам, як вимірювати власну продуктивність та співвідносити її з існуючим планом, і пояснює, чому описані методи можуть допомогти їм у їхній роботі. Передбачається, що РЗР допомагає виробити певні навички, особливо пов'язані з кількісною оцінкою роботи.

При цьому РЗР передбачає, що інженер вже має знання безпосередньо в галузі програмування.

Командний процес розробки програмного забезпечення (Т5Р)

У 1999 році Уоттс Хзмфрі представив позитивні результати у постановці цілей та процедур зрілості для командної розробки програм.

Перелічимо завдання ТЗР.

Зібрати самоврядні команди: 3-20 розробників;

встановити власні цілі;

скласти свій процес та плани;

відстежувати роботу.

Показати менеджерам. як управляти командами: інструктаж; мотивація;

підтримка максимальної продуктивності.

Прискорити просування за шкалою СММ:

зробити п'ятий рівень СММ нормою.

Забезпечити шляхи покращення для високорозвинених організацій.

Сприяти університетській освіті для команд промислового рівня.

Ставка ТЗР на командну ініціативу підтримує високий рівень професіоналізму серед програмних розробників.

Модель зрілості здібностей (СММ). СММ виявилася дуже афективною визначення рівня компетентності у створенні програмного забезпечення. Багато організацій, як військові, так і цивільні, використовували СММ для вимірювання своїх зусиль для покращення процесу розробки. Наприклад. замість того, щоб характеризувати організацію як «досить хорошу» у галузі розробки програмного забезпечення, можна точно вказати, що організація «знаходиться на рівні 3 за СММ.

СММ класифікує організації за п'ятьма рівнями в такий спосіб.

Рівень 1: Початковий. Рівень 1 відповідає найпримітивнішому статусу організації. На цьому рівні визнається, що організація може розробляти програмне забезпечення. Організація немає явно усвідомленого процесу, і якість продукту цілком визначається індивідуальними здібностями розробників. У звичайному випадку один з членів команди виявляє ініціативу - що робити далі, і команда слідує його вказівкам. Успіх одного проекту не гарантує успіху наступного (за винятком випадку, коли склад команди не змінюється і аналогічний проект). Після завершення проекту не фіксуються дані про трудовитрати. розклад та якість.

Процес: не визначено; залежить від конкретного завдання.

Результат залежить від індивідуальних здібностей.

Нестача: немає чітко заданого процесу.

Рівень 2: Повторюваний. Рівень 2 відповідає організаціям. Які певною мірою відстежують свій процес роботи. Підтримуються записи про трудовитрати та плани. Функціональність кожного проекту описана у письмовій формі. Отже, можна оцінити витрати на розробку аналогічного проекту тією самою командою. Це помітне покращення порівняно з рівнем 1. Для подальшого покращення необхідно мати можливість оцінювати витрати незалежно від наявності або відсутності конкретних людей у ​​команді проекту. У 1999 року лише 20 % організацій мали рівень 2 чи вище.

Процес: ведеться документація та облік трудовитрат. відстежується хід виконання планів та функціональності (постфактум).

Результат: відтворюємо лише для аналогічних проектів.

Недолік: немає повного процесу.

Рівень 3: встановлений. Рівень 3 відповідає організаціям, які мають певний, документований та встановлений процес роботи, який не залежить від окремих осіб. Зазвичай це один із процесів, описаних у цьому розділі: водопадний, спіральний або інкрементальний. Деякі організації використовують існуючі стандарти ШЕЕ, тоді як інші визначають власні корпоративні стандарти. Грубо кажучи, щойно керівництво вводить у дію узгоджені професійні стандарти, а розробники їх виконують, організація досягає рівня 3. Зазвичай це передбачає проведення спеціального навчання персоналу. Дозволяється налаштування стандартного процесу для конкретної команди стосовно конкретних обставин. Хоча організації рівня 3 в змозі виробляти програмне забезпечення гарантованої якості та таких організацій відносно небагато, їм все ж таки не вистачає можливості точно передбачати витрати на проект. Такі організації можуть досить надійно зробити прогноз тільки для проектів. які повністю аналогічні вже виконаним раніше.

Процес: документований; стандартний; налаштовується.

Результат: цілісність.

Недолік: результати не цілком передбачувані.

Рівень 4: Керований. Рівень 4 відповідає організаціям, які можуть точно передбачити терміни та вартість роботи. Один із способів, яким це робиться, полягає в тому, щоб класифікувати роботи та їх частини та вимірювати та зберігати дані про вартість та трудовитрати на проектування та реалізацію цих частин. Такі виміри накопичуються у базі даних, і основі робляться оцінки нових робіт.

# 2 управління проектом

П'ять етапів процесу планування програмного проекту.

Етап 1 Середа проекту, впливає на проект (внутрішні та зовнішні фактори)

Етап 2 Формулювання проекту - постановка цілей, завдань та вироблення стратегії реалізації проекту

Етап 3 Планування проекту – система заходів щодо реалізації проекту

Етап 4 Технічне виконання - безпосереднє технічне виконання пунктів плану проекту

Етап 5 Управління проектом - контроль за виконанням проекту відповідно до плану

Складові управління проектом.

Управління проектом охоплює:

♦ інфраструктуру (організаційні моменти);

♦ керуючий процес (права та відповідальності учасників);

♦ процес розробки (методи, інструменти, мови, документація та підтримка);

♦ розклад (моменти часу, до яких мають бути представлені виконані фрагменти роботи).

Основні параметри: вартість, функціональність, якість та розклад.

Планувальники проекту можуть варіювати вартість, можливості, якість та дату завершення проекту. Керівник проекту може керувати такими факторами.

1.

Загальна вартість проекту.

Наприклад, збільшувати витрати.

2.

Можливості продукту.

Наприклад, видаляти їх зі списку можливостей продукту.

3.

Якість продукту.

Наприклад, збільшувати середній час напрацювання на відмову (MTBF).

4.

Тривалість проекту.

Наприклад, скоротити розклад на 20% або відкласти завершення проекту на один місяць.

Ступінь контролю за цими чотирма чинниками залежить від проекту. Незважаючи на те, що вартість може бути обумовленою заздалегідь та фіксованою, найчастіше існують різні гнучкі варіанти.

Розподіл обов'язків між трьома досвідченими інженерами при виконанні роботи, що вимагає три людино-місяця.

Грунтуючись на тому, що учасники команди зацікавлені в проекті, розумно використовувати модель команди рівних, виділивши одну людину як лідер проекту. Він задаватиме тон, прийматиме вольові рішення і слідкуватиме за розкладом. Інші беруть він кожен певну частину процесу.

Кажуть, що «не буває технічних провалів, а бувають лише провали в управлінні». Хоча це не зовсім правильно, проте нагадує нам про важливість управління для успішного завершення діяльності з розробки. Ставлення менеджера до своїх підлеглих - це суміш ділового інтересу з інтересом роботи з людьми, що беруть участь у проекті. Незважаючи на часті побоювання колективу, середньостатистичний менеджер намагається довести роботу до кінця так, щоб у підлеглих не виникало почуття невдоволення чи незадоволеності. Менеджер сам у цьому зацікавлений, оскільки незадоволені працівники працюють непродуктивно. Однією з найважчих завдань менеджера є досягнення головної мети виконання роботи за умови отримання колективом задоволення від виконаної роботи. Наприклад, вище керівництво може поставити умову,

Менеджер повинен зробити так, щоб технічні дослідження розробників мали потрібний напрямок. Менеджери-початківці часто знаходять досить складним лавірування між відданням наказів і наданням свободи розробникам. Диктатура в управлінні може викликати обурення колективу та призвести до втрати мотивації. Однак надмірне потурання може призвести до втрати часу і марно виконаної роботи. Вирішенням цієї проблеми є лідерство, з'ясування істинних бажань та потреб людей, активне їх розподіл та об'єднання у спробі досягнення успіху. Лідери проекту повинні варіювати ступінь своєї відповідальності в управлінні, ґрунтуючись на величині проекту. У великих проектах їхні обов'язки перебувають у основному управлінні. У малих проектах лідери повинні забезпечувати як загальне управління,

Проект такого розміру потребує помітних зусиль на організацію. Повинна бути встановлена ​​ієрархія управління (принаймні керівник проекту/керівники за напрямками/розробниками). Щодо невелика команда кращих розробників має бути призначена на підготовку та аналіз вимог та розробку архітектури. Робота має бути розбита на кілька частин, і керівники за напрямами повинні вести кожну частину, на кожну частину можна призначити своїх розробників. Керівники напрямів у підрозділах можуть використовувати модель команди рівних.

Без письмового плану навіть найкомпетентніші розробники будуть марнувати час і виконувати порожню роботу. Незалежно від того, написаний план на папері чи ні, він все одно існує, можливо в головах деяких розробників. Якщо план не написаний, він має бути промовлено усно. Хоча це звучить сильно, насправді це дуже непрофесійний підхід, і малоймовірно, що в результаті вийде професійний продукт. Усна передача інформації тим погана, що втрачаються такі речі:.

♦ який процес розробки використовується;

♦ хто, що і коли має робити;

♦ які ризики загрожують у майбутньому. Без плану ідентифікації та усунення ризиків проект здається на милість невідомих обставин у майбутньому.

Можна уявити чернетку плану-графіка, але в ньому неможливо передбачити деталі. Справа в тому, що детальний план можна скласти лише знаючи, як робота поділена на частини, а це невідомо на даний момент, бо ще не проведено проектування архітектури.

План проекту має бути складено так, щоб кожен знав, що і коли йому потрібно робити. Існує багато стандартів для таких планів. Ми будемо використовувати стандарт IEEE 1058.1-1987 (затверджений у 1993 році). План управління програмним проектом (SPMP — Software Management Plan).

Будь-який проект має ризики, пов'язані з вартістю, розкладом та якістю. Рання ідентифікація ризиків залишає час на вжиття заходів щодо їх попередження, що зменшує збитки від несприятливих подій.

Огляд керування ризиками.

Програми, схожі на ті, які були розроблені раніше тими самими інженерами, зазвичай не є ризикованими. Однак різноманітність програмних додатків величезна та швидко зростає. Багато робіт містять нові шляхи представлення задач або є реалізаціями нових ідей. З цих причин у розробці програмних додатків зазвичай є багато ризиків.

Кожен ідентифікований ризик повинен з радістю сприйматися командою проекту, тому що в цьому випадку з ним можна щось робити. Справжньою проблемою є ризики, які вдалося ідентифікувати. Такі ризики схожі на міни, які чекають на свої цілі. Оскільки відсоток проектів, які вже ніколи не завершаться, воістину величезний, то постійна увага до ризиків робить вірогіднішим той факт, що проект потрапить у 20 % більш удачливих проектів або буде припинено до того, як будуть витрачені величезні гроші та занапащені кар'єри інженерів.

Управління ризиком складається з кількох дій.

1. Ідентифікація.

Намагайтеся постійно виявляти ризики.

2. Планування усунення.

3.Вибір пріоритетів.

4.Усунення або зменшення.

Ці кроки потрібно здійснювати від початку проекту та вперто продовжувати протягом першої чверті проекту. Деякі команди призначають одного зі своїх колег на роль координатора ризиків, відповідального за пошук ризиків, а також інформує їх усунення.

Може статися так, що в процесі ідентифікації ризиків вдасться виявити лише малу їх кількість, причому таких, для яких заходи щодо запобігання було б вжито в будь-якому випадку. У цьому випадку час виявиться витрачений дарма, і єдиним корисним результатом буде заспокоєння.

Попередження ризиків - це процес, під час якого рівень ризиків знижується або ризики повністю усуваються. Є два способи запобігання ризикам. Перший полягає у внесенні змін до вимог проекту, завдяки чому усувається причина виникнення ризику (уникнення ризику). Інший метод - розробка деяких технологій і архітектури, що вирішують проблему (подолання ризику або, простіше кажучи, його усунення). Попередження ризиків можна порівняти з прокладанням маршруту пішохода в обхід перешкод вулиці та будинки.

Процес оцінки вартості (для фіксованих можливостей, рівня якості та розкладу) часто починається на початку проекту і продовжується навіть на стадії написання програмного коду. При відкритті проекту команда може мати дуже розпливчасте уявлення про його вартість. Якщо оцінку вартості можна відкласти до тих пір, поки проект не набере повного ходу, то так і потрібно зробити, але завжди є потреба хоча б грубої оцінки діапазону вартості вже на фазі аналізу вимог. Чим більше ми знаємо про вимоги до продукту, і чим далі просунуто проектування, тим точніше ми можемо оцінити вартість проекту.

Може виникнути спокуса відповісти на це питання так: невеликий проект, який потрібно виконати за всяку ціну. Потрібно погодитись, що час, витрачений на оцінку, забирає надто велику частку з часу, відпущеного на роботу. Але це не виправдовує відмову від оцінки вартості. Можна вважати, що час, необхідний оцінки обсягу роботи, пропорційно обсягу роботи, отже обсяг проекту немає великого значення. Відмова від оцінки може бути виправдана, тільки якщо той, хто оцінює, зовсім не вміє цього робити. Справді, якщо три інженери два тижні вчаться оцінювати вартість проектів, а потім застосовують отримані знання, щоб оцінити один двомісячний проект, то гра не варта свічок. У великих проектах ці два тижні трохи важать.

Метод виміру функціонального розміру полягає у одноманітному вимірі всіх можливостей докладання та вираженні розміру додатку у вигляді одного числа. Це число можна далі використовувати для оцінки кількості рядків коду, вартості та термінів проекту. Функціональний розмір - це дуже привабливе поняття, оскільки воно претендує на те, щоб виміряти суть можливостей майбутньої програми. Однак потрібно мати певну навичку, щоб застосовувати цей метод акуратно та послідовно.

Перевага: можливість зробити дуже ранню оцінку вартості. Недолік: потенційно високий розкид у значенні функціонального розміру, особливо якщо застосовуючий не має достатнього досвіду.

У моделі Боема стверджується, передусім, що з різних типів додатків тривалість і трудовитрати по-різному залежить від розміру додатків (відрізняються множником і показником експоненти).

Переваги: ​​можливість оцінити одночасно терміни та трудовитрати; точність формул. Недолік: метод заснований на вимірі кількості рядків коду, а ця величина часто невідома.

Метрики процесу вимірюють ефективність процесу. Наприклад, ми можемо порівняти ефективність водопадної та спіральної моделі для даної організації на даний час. Наступні метрики дозволяють це зробити.

♦ (Загальний час проектування)/(Загальний час розробки). Зазвичай має бути 0,5 або трохи більше.

♦ (Кількість дефектів на тисячу рядків)/(Середнє по компанії).

♦ (Частка недокументованих вимог)/(Середнє за компанією).

♦ [(Фактична тривалість проекту)/(Планова тривалість проекту)]/(Середнє по компанії).

Останні три метрики є метриками процесу і можуть бути окремі проекти, які мають значні відхилення від середнього.

# 3 Аналіз вимог

Набуття конкретних вимог – складний процес. Він складається з чуйної взаємодії з тими, хто фінансово зацікавлений у успіху програмного додатку.

Протягом деякого часу проходили дебати щодо того, кому «належать» вимоги: замовнику чи розробникам. Результати замовника іноді називають вимогами замовника, або

З вимогами. Первинною аудиторією для С-вимог буде співтовариство замовників, а вже вторинною — спільнота розробників. Другий рівень документує вимоги у спеціальній, структурованій формі. Ці документи називаються вимогами розробника, або

D-вимоги. Первинною аудиторією для D-вимог буде співтовариство розробників, а вже вторинною — спільнота замовників.

С-вимоги виражають вимоги у формі, зручній замовнику, і складаються переважно з високорівневого опису. Форма D-вимог зручна для розробників. D-вимоги є більш докладною формою С-вимог.

Переваги аналізу вимог. Недосконалі вимоги коштують дуже дорого. За оцінками, їх у 20-50 разів дорожче виправити, якщо вони потрапили у процес розробки. Збитки, пов'язані з негативним досвідом роботи замовника з додатком, є додатковим чинником оцінки витрат.

Чому ж так багато проектів зазнало збитків від мізерного чи не проведеного аналізу вимог, якщо відомий значний прибуток від виявлення помилок під час формулювання? Основна причина полягає в тому, що замовники зазвичай не знають на початку проекту, чого вони хочуть або чого потребують. Вивчення ринку відеоігор наприкінці лекції є прикладом цієї невпевненості: проект має сформульовану мету, але його зміст усе ще змінюється. У цій книзі робиться акцент на ітераційну розробку та ретельну перевірку вимог, проектування та реалізації. Інженери, які використовують добре організований ітераційний процес, збирають вимоги, проводять проектування та виконують реалізацію узгодженими ітераціями.

Переваги та недоліки поділу вимог на категорії. Замовники та розробники мають різні потреби у вимогах. До переваг поділу С-і D-вимог відноситься той факт, що вони швидше задовольнять ці різні потреби. До недоліків відноситься можливість того, що дві форми опису не співпадатимуть.

**Приклад побажань замовника.**

*Результат С-вимог для гри «Зустріч»*

 *- Гра моделює усі сторони життя персонажа.*

 *- Ігрові персонажі, які не контролюються гравцем, називаються зовнішніми персонажами*

 *- У кожного набору персонажів є набір характеристик*

 *- Кожна характеристика має чисельне значення*

 *- Персонажі трапляються, якщо знаходяться в одній зоні*

 *- Результат зустрічі залежить від характеристик та зони*

 *- Персонаж може перерозподіляти свої характеристики*

 *- Перерозподіл набирає чинності через деякий час*

 *- Успіх визначається за одним із принципів:*

 *- максимальна кількість очок-життя*

 *- виживання протягом якомога більш тривалого періоду.*

Вимоги часто, природно, виражають через взаємодію програми із зовнішнім користувачем. Варіанти використання є дуже корисним способом висловлювання вимог замовника у вигляді таких взаємодій. Варіант використання – це послідовність взаємодії користувача та програми у типовій ситуації.

Варіант використання визначається, перш за все, своїм ім'ям та типом користувача програми, що називається дійовою особою.

Варіант використання складається з типової взаємодії між дійовою особою та додатком.

*«Система буде надавати пораду початківцю WINDOWS щодо того, як слід виконувати операції WINDOWS»:*- це не варіант використання, оскільки тут не продемонстрована послідовність дій, виконаних програмою та користувачем програми.

У розробці інтерфейсів для наших прикладних завдань удачливі з нас мають можливість працювати з професійним дизайнером, або мати хоча б його підтримку. Однак для багатьох проектів, особливо невеликих, розробники програмного продукту повинні розробляти дизайн інтерфейсу користувача самі. Тому перерахуємо основні кроки розробки інтерфейсу користувача:

1)Дізнатися свого користувача

2) Зрозумійте призначення проектованої системи

3) Застосуйте принцип гарного екранного дизайну

4) Підберіть відповідний тип вікна

5) Розробте системне меню

6) Виберіть відповідні апаратні пристрої керування

7) Виберіть відповідні екранні елементи керування

8) Організуйте та створіть розкладку вікон

9) Виберіть відповідні кольори

10) Створіть осмислені значки

11) Надайте ефективні повідомлення, зворотний зв'язок та керівництво.

**Швидке прототипування**– це часткова реалізація цільової програми, у тому числі значної частини інтерфейсу користувача. Чим детальніше прототип, тим легше зрозуміти вимогу замовника. З іншого боку, прототипи самі по собі є програмами, тому чим докладніший прототип, тим він дорожчий.

**Як приклад:**

*Має сенс створити закінчений прототип для відстеження посилок у великій поштовій компанії.*

*Для простого зберігання інформації CD-дисків створення прототипу недоцільно, занадто мала програма, щоб відправляти прототип.*

*Для підтримки пенсійних рахунків створення закінченого прототипу можливе (ні, якщо програма передбачається для одного замовника; можливо, якщо передбачається подавати її як продукт багатьом замовникам).*

Іноді не очевидно, чи взагалі можуть бути реалізовані запропоновані вимоги. Іншими словами, існує ризик для всього проекту, а не ризик щодо конкретних вимог. У таких випадках, можливо, потрібно буде провести дослідження здійсненності вимог. Дослідження здійсненності є частковою розробкою або моделюванням програми.

Приклад програми, що вимагають дослідження здійсненності:

*1. Система без даних: мабуть, надто стандартно, щоб проводити дослідження здійсненності.*

*2.Автоматичне резюме: так*

# 4 Аналіз вимог. Завершення SRS: додавання детальних вимог.

Детальні вимоги (D-вимоги) є єдиним документом, в якому визначається конкретна природа програми. Рівень деталізації має бути повним, але не надмірним.

Розробникам програмного забезпечення потрібна база для проектування та розробки. Ця база складається із детальних вимог. Їх також називають конкретними вимогами, функціональними специфікаціями, вимогами розробника чи D-вимогами. D-вимоги складаються з повного списку конкретних властивостей та функціональності, яку повинна мати програма, сформульована у подробицях. Кожна з цих вимог пронумерована, позначена і відстежується в процесі розробки. D-вимоги повинні бути узгоджені із С-вимогами.

Передбачається, що D-вимоги читатимуть переважно розробники. Замовники також зацікавлені у них і зазвичай можуть зрозуміти чи прокоментувати більшість із них. Основна аудиторія для С-вимог складається із замовників.

Типова послідовність дій для збору та документування D-вимог:

Вибрати систему організації D-вимог

Створити діаграми послідовності для варіантів використання

3.1. Отримати D-вимоги із С-вимог та від замовника

3.2. Зробити чернетку планів тестування

3.3. Провести інспектування

4. Затвердити із замовником

5. Випустити

Типи D-вимог

Функціональні вимоги:

- функціональність програми

2. Нефункціональні вимоги

2.1. Продуктивність:

 -швидкість

 -пропускна спроможність

 -використання пам'яті

2.2. Надійність та доступність

2.3. Обробка помилок

2.4. Інтерфейсні вимоги

- як програма взаємодіє з користувачем та іншими програмами

2.5. Обмеженість

 - точність

 - обмеження на інструменти та мову

 - обмеження проектування

 - стандарти, які мають бути використані

 - платформи, які мають бути використані.

3. Зворотні вимоги

Проблеми, що супроводжують створення вимог та роботи з ними, варто класифікувати так, щоб їх можна було легко знаходити та підтримувати.

Категорії детальних вимог: функціональні, нефункціональні, зворотні, інтерфейсні, проектні та обмеження реалізації.

*-Функціональні*вимоги визначають роботу, яку має виконувати програмний додаток (наприклад, «додаток обчислюватиме вартість портфеля акцій користувача»).

*-Нефункціональні вимоги: вимоги до продуктивності:*

є важливою частиною додатків, що працюють у реальному часі, у яких дії повинні вкластися у визначені часові рамки. Прикладами додатків реального часу можуть бути програми уникнення зіткнень, контролю польотів та антиблокувальної гальмівної системи.

*-Нефункціональні вимоги: надійність та доступність*.

Вимоги надійності визначають надійність у вимірюваних величинах. Вимоги такого типу припускають можливість неідеальної роботи програми і обмежують область її недосконалості.

Доступність, близька за змістом надійності, оцінює ступінь, в якій програма має бути доступна користувачам.

*-Нефункціональні вимоги: обробка помилок.*

Ця категорія вимог пояснює, як програма повинна реагувати на помилки, що виникають. Цей тип вимог до помилок має бути використаний вибірково, оскільки нашою метою є створення безпомилкових програм, а не виправлення помилок за допомогою нескінченного коду обробки помилок.

*-нефункціональні вимоги: інтерфейсні вимоги.*

Інтерфейсні вимоги описують формат, у якому програма спілкується із оточенням.

Вартість посилки статті від адресата одержувачу повинна постійно відображатись у текстовому вікні «Ціна».

*-Нефункціональні вимоги: обмеження.*

Обмеження на проектування або реалізацію визначає межі або умови того, як додаток має бути задуманим та розробленим. Ці вимоги не повинні використовуватися замість процесу проектування — вони лише визначають умови, накладені на проект замовником, а також оточення чи інші обставини.

*-Зворотні вимоги.*

Зворотні вимоги визначають, чого програма робитиме. Логічно, що існує безліч зворотних вимог: ми вибираємо тільки ті, які роз'яснюють можливе нерозуміння.

Бажані властивості D-вимог: простежуваність, тестування, однозначність, пріоритет, повнота, умови помилок, узгодженість.

*-Простеження функціональних вимог.*

Можливість відображати кожну вимогу на відповідні частини проекту та програми називається простежуванням. Один із способів, що допомагають досягти цього, полягає у відображенні кожної функціональної D-вимоги на конкретну функцію цільової мови.

*-Придатність до тестування та однозначність.*

Повинна бути можливість валідувати вимогу, переконавшись, що вимога була коректно реалізована. Вимоги, які можна протестувати, називають тестованими. Нетестовані «вимоги» не мають великої практичної цінності.

*-Пріоритет.*

Часто буває важко реалізувати всю заплановану функціональність програми вчасно і не виходячи за межі бюджету. Призначення пріоритетів вимогам позначається на проектуванні, оскільки бажані та необов'язкові вимоги часто вказують напрямок, у якому має рухатися програма. Широко відома мудрість говорить, що підприємства отримують близько 80% доходу від використання лише 20% вимог. Тому, якщо пріоритети розставлені правильно (наприклад, лише близько 20% вимог віднесено до суттєвих), можна отримати максимальний прибуток від програми з урахуванням необхідної роботи. Цей фактор слід враховувати, якщо проект починає випадати з тимчасових рамок.

*-Повнота.*

Повнота набору вимог гарантує, що не було зроблено жодних перепусток, що підривають сформульовані вимоги.

*-Стан помилки.*

Для кожної вимоги потрібно порушити питання: «Що буде, якщо станеться помилка?». Це один з еквівалентів практики програмування, що давно склалася, коли надмірність використовується для підвищення надійності.

*-Узгодженість.*

Набір D-вимог узгоджено, якщо між вимогами немає протиріч. У міру збільшення числа D-вимог узгодженість може стати важкодосяжною.

Способи організації детальних вимог.

D-вимоги можна організувати за допомогою кількох схем:

*-За основними властивостями*. Цей метод організації часто сприймають як «вимоги», маючи на увазі, що вимоги згруповані за різними властивостями програми. Саме собою це не надає жодної систематичної організації, оскільки дозволяє переходити від якості однієї частини програми до якості абсолютно іншої частини програми.

*-за режимом*(наприклад, системи управління радарами можуть мати тренувальний, нормальний та аварійний режими).

*-за варіантами використання*(Іноді ще називається за сценаріями). Ідея полягає в тому, що більшість детальних вимог є частиною варіантів використання.

*-за класом.*Це об'єктно-орієнтований стиль. У цьому вся способі організації класифікуються вимоги по класам.

*-по ієрархії функцій*(тобто шляхом розбиття програми на безліч високорівневих функцій та подальшого розбиття їх на підфункції тощо).

*-за станами*(тобто шляхом зазначення детальних вимог, які застосовуються до кожного стану). Зокрема, вимоги для програми, що управляє хімічним процесом, краще класифікувати за станами, в яких може знаходитися процес (початок, реакція, охолодження і т. д.).

Класифікація за станами може бути доречною, якщо вимоги для кожного стану сильно відрізняються.

Діаграма послідовності. Завжди можна визначити торкнутіся у варіанті використання об'єкти, а потім перетворити послідовність дій користувача (системи) на послідовність виклику функцій цих об'єктів.

Не існує варіант використання, який відповідає кожній діаграмі послідовності. Діаграма послідовності виражає послідовність функцій викликів. Не кожна така послідовність є типовою взаємодією між користувачем і програмою.

Існує варіант використання, який відповідає кожній діаграмі послідовності.

Отже, D-вимоги (вимоги розробників чи детальні вимоги) пишуться переважно для проектувальників та розробників. Вони створюються із С-вимог, а також внаслідок тривалих переговорів із замовником. D-вимоги повинні бути тестованими, простежуваними та узгодженими. Оскільки вони розростаються до більших обсягів, слід систематично класифікувати. Зручним способом організації D-вимог є класифікація за основними класами предметної області та функцій. Існує кілька метрик, за якими можна перевірити D-вимоги. Хороший аналіз вимог приносить значну користь. Підіб'ємо підсумки цього розділу:.

- D-вимоги призначені для розробників.

- Цілі складання D-вимог: ясність, простежуваність вимог.

- Хороша організація приносить велику користь. Приклад – об'єктно-орієнтований стиль.

- Слід по можливості використовувати формальні методи.

– Результатом є оновлення SPMP.

# 6 Детальне проектування

## 6.1. Введення у детальне проектування

6.1.1 Що таке детальне проектування?

Детальне проектування— це технічна діяльність, яка слідує за вибором архітектури. Основною метою цієї діяльності є якомога повна підготовка проекту до його реалізації. Іншими словами, програмісти повинні мати можливість реалізувати детальний проект, сконцентрувавшись лише на проблемах, пов'язаних із створенням програмного коду.

6.1.2 Співвідношення варіантів використання архітектури та детального проектування

У проектуванні програмного забезпечення кожен наступний рівень накопичує додаткові класи:

Крок 1 – варіанти використання фіксуються як частина вимог.

Крок 2 - вони разом з іншими джерелами використовуються визначення класів предметної області.

Крок 3 – розробляємо програмну архітектуру (див. розділ 5).

Крок4 - перевірка того факту, що архітектура та детальний проект задовольняють необхідним варіантам використання.

ПрикладРозглянемо для прикладу завдання проектування моста.



Рисунок 6.1.2 Зв'язок між варіантами використання, архітектурою та детальним проектуванням

Стосовно нашого прикладу з проектуванням мосту ми перевіряємо те, що автомобілі дійсно зможуть використати спроектований міст під час подорожі з Пагорба Сміта до Долини Джонса, як це було заявлено у варіантах використання. У разі програмного проекту ми перевіряємо те, що класи та методи, специфіковані під час детального проектування, здатні реалізувати необхідні варіанти використання.

6.1.3. Типова схема процесу детального проектування

Детальне проектування починається за результатами фази архітектури і завершується з отримання повного плану для фази програмування.

Типова схема детального проектування:

1-й етап - проектування архітектурних моделей

• Модель класів: класи предметної галузі та архітектури

• Загальна модель переходів станів

• Загальна модель потоків даних

• Модель варіантів використання

2й етап- Представте класи та зразки проектування, які включають класи архітектури та предметної області.

3й етап - Удосконалюйте моделі, забезпечуйте несуперечність, стежте за їхньою повнотою

4-й етап - Визначте інваріанти класу для кожного методу.

5й етап - Використовуйте перед- та постумови, блок-схеми та псевдокод для кожної компоненти

6-й етап - Накиньте план модульного тестування

7й етап - Проінспектуйте плани тестування та проектування

8й етап - Запускайте у реалізацію

6.1.4. Проектування за схемою USDP (USDP – уніфікований процес розробки ПЗ)

У USDP детальне проектування має місце переважно під час ітерацій проектування та конструювання. USDP підтримує три стереотипи класів на рівні аналізу, не пов'язаних із класами проектування: класи сутності, граничні класи та класи управління.

Проектування з використанням інтерфейсів можна порівняти з укладанням контракту, причому елементи програми, що використовують цю основу функціональності, слід проектувати так, щоб вони не знали подробиці реалізації функціональності.

p align="justify"> Наступним прийомом детального проектування є повторно використовувані компоненти, розвитку якого сприяло широке поширення об'єктно - орієнтованих, об'єктно-подібних та інших парадигм компонентів. Прикладом повторного використання програмного коду є застосування бібліотеки Microsoft MFC, елементів керування Visual Basic, COM, Java Beans та інших класів Java API. Стандартом для розподіленого використання є архітектура CORBA консорціуму OMG. Веб-додатки (не компоненти), також часто бувають повторно використовуваними, і, нарешті, STL - бібліотека стандартних шаблонів, застосовується до різних структур даних та об'єктів практично будь-яких класів.

USDP підтримує три типи (стереотипи) класів на рівні аналізу, не пов'язаних із класами проектування: класи сутності, граничні класи та класи управління. У класах сутності виражена суть концепції проекту, та його зміни у додатках небажані. Граничні класи керують взаємодією між об'єктами сутності, а класи управління містять методи, що належать цим об'єктам, причому їх наявність особливо притаманна додатків, у яких використовуються класи сутності.

6.1.5. Проектування та інтерфейси

Проектування з допомогою інтерфейсів можна порівняти з укладанні договору. Елемент програми (наприклад, клас Замовник), що забезпечує необхідну функціональність, гарантує надання функцій з певними іменами, типами параметрів та значень, що повертаються.

Елементи додатки, що використовують цю основу функціональності, слід проектувати так, щоб вони не знали подробиць реалізації функціональності.

Застосування інтерфейсів у проектуванні може мати різні форми. Одна з них це абстракція.

Допустимо, ми пишемо код програми для роботи із замовниками. Це можна розуміти як написання коду на основі інтерфейсу Замовник. Взагалі кажучи, ми можемо розглядати абстрактний клас Замовник із неабстрактним підкласом, наприклад, Постійний Замовник (рис. 6.1.5). Подібний підхід до проектування більш гнучкий, ніж робота з конкретним (не абстрактним) класом Замовник, тому він дозволяє нам легко додавати нові типи замовників з їх індивідуальними версіями методу рахунок (), і при цьому нам не потрібно змінювати код, що маніпулює об'єктами Замовник. Такий поділ на абстрактний та конкретний рівні характерний для багатьох зразків проектування.



Рисунок 6.1.5 Проектування та інтерфейси

6.1.6. Повторно використовувані компоненти

Більшість інженерних дисциплін (електрика, механіка і т. д.) засновані на використанні компонентів, які можуть бути виконані окремо, незалежно від інших компонентів. Наприклад, проектувальники мостів, взагалі кажучи, не займаються проектуванням конструкцій балок. З року в рік спостерігається та сама ситуація: програмні розробники безперервно «винаходять велосипед», реалізуючи вже давно існуючі процедури. Однією з причин недостатнього поширення повторного використання програмного забезпечення є невдала організація існуючих архітектур, проектів і коду. Розвитку повторного використання програмного забезпечення сприяло широке поширення об'єктно-орієнтованих, об'єктно-подібних та інших парадигм компонентів.

Завдяки величезній кількості методів, що надаються в пакеті з кожним класом, потрібна нам функціональність часто вже включена і доступна.

Прикладом повторного використання програмного коду є застосування бібліотеки Microsoft MFC, елементів керування Visual Basic, COM, Java Beans та інших класів Java API. Стандартом для розподіленого використання є архітектура CORBA консорціуму OMG.

Комбіновані можливості застосування стандартних алгоритмів, наприклад сортування і пошуку, можуть надати бібліотеки стандартних шаблонів (STL). STL застосовуються до різних структур даних та об'єктів практично будь-яких класів.

У цілому нині слід зазначити, що ринок компонентів вже сформувався і розвивається.

## 6.2. Діаграми послідовності та діаграми потоків даних у детальному проектуванні

Діаграми послідовностікорисні у тому, щоб зрозуміти логічну послідовність подій у сценарії. Діаграми послідовності – це основний інструмент прийняття рішень про розподіл поведінки.

Діаграми послідовності

1. Почніть із діаграм послідовності, створених для детальних вимог або архітектури, що стосуються варіантів використання.

2. Якщо це необхідно, представте додаткові варіанти використання для показу взаємодії проектованих частин з рештою програми.

3. Забезпечте повну деталізацію діаграм послідовності.

переконайтеся, що всі об'єкти та їх класи специфіковані;

виберіть конкретні імена функцій замість природної мови для представлення операцій.

Діаграми потоків даних (data flow diagram, DFD) - один з основних інструментів структурного аналізу та проектування інформаційних систем, що існували до поширення[UML](https://ru.wikipedia.org/wiki/UML). Незважаючи на наявне в сучасних умовах зсув акцентів від структурного до об'єктно-орієнтованого підходу до аналізу та проектування систем, «старовинні» структурні нотації, як і раніше, широко і ефективно використовуються як в бізнес-аналізі, так і в аналізі інформаційних систем.

Діаграми потоків даних

1. Зберіть діаграми потоків даних (ДПД), створені для детальних вимог та (або) архітектури.

2. За потреби, наведіть додаткові діаграми потоків даних, щоб пояснити потоки даних та процес.

3. З'ясуйте, яким частинам інших моделей відповідають діаграми потоків даних

4. Забезпечте повну деталізацію діаграм потоків даних:

З'ясуйте природу процесу, що відбувається у кожному вузлі;

З'ясуйте тип даних, що передаються;

Якщо опис процесу потребує більшої деталізації, розкрийте вузли процесу у діаграмі.

## 6.3. Специфікація класів та функцій

Метою детального проектування є повноцінний проект, яким може бути створена програма. Добре побудований проект залишає творця з мінімальною кількістю невизначеностей щодо намірів розробника. Це слушно і для програмного проектування.

Типові кроки при специфікації класу:

1. Зберіть атрибути, наведені в SRS. Це легше зробити, якщо SRS організована за класами.

2. Додайте додаткові атрибути, які необхідні для проектування.

3. Назвіть методи, які відповідають кожному з вимог цього класу. Це легше зробити, якщо SRS організована за класами.

4. Назвіть додаткові способи, необхідні для проектування.

5. Подайте атрибути та методи на об'єктній моделі.

6. Встановіть інваріанти класу.

Типові кроки при специфікації функції:

1. Позначте розділи SRS або SDD, які задовольняють цю функцію (метод).

2. Визначте, у яких виразах функція має бути інваріантною.

3. Встановіть передумови методу (за яких умов виконується).

4. Задайте постумови методу (що відбувається внаслідок його виконання).

5. Забезпечте опис алгоритму, що використовується у вигляді псевдокоду або блок-схеми.

Винятком є ​​випадок, коли алгоритм є тривіальним.

6.3.1. Інваріанти класу

Інваріант - це твердження, що залишається справедливим протягом усього обчислень. Добре продумані класи 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 програми, і тому можуть увійти до SRS. Інакше вони пред'являються під час проектування. Інваріанти класу набувають форми обмежень на значення. Як і вимоги, вони часто називаються правилами бізнесу. Наприклад, вимога "Всі учасники веб-аукціонів повинні надавати номер своєї кредитної картки"

6.3.2. Інваріанти, передумови та постумови функцій

Один із способів управління роботою функцій ґрунтується на застосуванні інваріантів функцій. Так називаються твердження про взаємозв'язки змінних, які гарантовано задовольняються функціями.

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

## 6.4. Специфікація алгоритмів

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

6.4.1. Блок-схеми

Блок-схеми є одним із найстаріших графічних уявлень алгоритмів. Приклад блок-схеми для методу setName() класу GameCharacter (ПерсонажІгри), що демонструє два найпростіші елементи блок-схем - умови (ромби) та дії (прямокутники).

Завдяки інспекцію блок-схеми на рис. 6.4.1 ми можемо побачити, що перевірка допустимості імені параметра підпорядковується досить туманною логікою.


Малюнок 6.4.1. Приклад блок-схеми

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

Використання блок-схем знизилося за останні два десятиліття двадцятого століття. Частково це сталося через поширення об'єктно-орієнтованого підходу, який став досить популярним саме в ті роки у зв'язку з тенденцією скорочення кількості розгалужень завдяки розподілу функціональності за класами та використанню віртуальних функцій.

6.4.2. Псевдокод

Псевдокод - це варіант текстового представлення алгоритму, що дозволяє не вдаватися до деталей мов програмування. Основною перевагою псевдокода є його зрозумілість та можливість досить точно уявити алгоритм. Друга перевага полягає у можливості інспектування алгоритмів щодо їх коректності незалежно від нагромаджень мови програмування. Третя перевага в тому, що ми можемо накопичувати статистику по відсотку помилок у псевдокоді, а потім використовувати ці дані для передбачення кількості дефектів у кінцевому продукті. Багато організацій використовують інспектований псевдокод як коментарі у лістингу вихідного програмного коду.

6.5. Зразки проектування: прийоми детального проектування

Зразки проектування повинні використовуватися тільки в сприятливих умовах, оскільки в іншому випадку вони можуть ускладнити тільки звичайне проектування. Незважаючи на ці застереження, автор щиро вірить у те, що зразки проектування є невід'ємною частиною інструментарію програмних розробників.

Кожен зразок проектування має два аспекти. Перший аспект - це звичайна модель класів, що описує класи та їх взаємозв'язок. Другий аспект є способом дії зразка проектування.

Зразки проектування поділяються на три категорії:

1. Креаційні зразки проектування, відповідальні за створення об'єктів.

2. Структурні зразки проектування, що займаються відображенням можливих комбінацій об'єктів.

3. Поведінкові зразки проектування – фіксує вибраний режим стану об'єктів.

## 6.6. Бібліотека стандартних шаблонів (STL) C++

Бібліотека стандартних шаблонів (STL — Standard Template Library), створена А. Степановим, є бібліотекою стандартних структур даних і алгоритмів, що багаторазово використовуються, заснованої на використанні шаблонів. STL – це частина стандарту ANSI C++.

Основні складові STL: контейнери. Містить колекції об'єктів: вектори, списки, стеки та черги.

♦ алгоритми. Алгоритми маніпулюють STL-контейнерами і включають: вставку, видалення, пошук і сортування. ♦ ітератори. Ітератори представляють операцію над елементами контейнера. ♦ функціональні об'єкти. Кожен функціональний об'єкт приймає аргумент і повертає результат, але при цьому він залишається вихідним об'єктом. До функціональних об'єктів відносяться такі операції: розподіл, логічне І, «Не одно» тощо.

♦ адаптери.

6.7. Стандарти, нотація та інструментальні засоби детального проектування

6.7.1. Стандарт IEEE 890

Для об'єктно-орієнтованого проектування послідовного опису кожного модуля з детальним описом кожного блоку даних може бути замінено детальним описом кожного класу.

1. Введення

1.1. Ціль

1.2. Опис проекту

1.3. Визначення, скорочення та терміни

2. Посилання

3. Опис декомпозиції

3.1. Модульна декомпозиція

3.1.1. Опис модуля 1

3.1.2. Опис модуля 2

3.2. Декомпозиція на паралельні процеси

3.2.1. Опис процесу 1

3.2.2. Опис процесу 2

3.3. Декомпозиція даних

3.3.1. Опис блоку даних 1

3.3.2. Опис блоку даних 2

4. Опис залежностей

4.1. Міжмодульні залежності

4.2. Міжпроцесні залежності

4.3. Залежності всередині даних

5. Опис інтерфейсу

5.1. Модульний інтерфейс

5.1.1. Опис модуля 1

5.1.2. Опис модуля 2

5.2. Інтерфейс процесів

5.2.1. Опис процесу 1

5.2.2. Опис процесу 2

6. Детальне проектування

6.1. Детальне проектування модулів

6.1.1. Модуль 1: деталі

6.1.2. Модуль 2: деталі

6.2. Детальне проектування даних

6.2.1. Блок даних 1: деталі

6.2.2. Блок даних 2: деталі

6.7.2. Мова UML

UML дозволяє представити всі методи у детальному проектуванні. Оскільки класи, як правило, містять безліч методів, часто бажано їх вибіркове уявлення. Це стало можливим завдяки інструментам Rational Rose та Together/J. Вони також значно полегшують масштабування об'єктної моделі.

6.7.3. Інструменти, що використовують вихідний код: Javadoc

Діаграми ієрархії класів, що використовують гіперпосилання, можуть бути згенеровані безпосередньо з вихідного коду. Наприклад, Javadoc генерує списки класів, що належать пакетам, та методів, що належать до класів. З коментарів у вихідному коді, оформлених спеціальним чином, Javadoc також генерує опис цих компонентів. Такі інструменти як Javadoc реалізують так зване зворотне проектування. Це означає, що вся документація на продукт (включаючи проект) проводиться після, а не до створення продукту. В принципі, в будь-якому проектуванні певною мірою є зворотне проектування. Зокрема, програмісти зазвичай додають методи, які не передбачені при детальному проектуванні. Однак зворотне проектування неефективне, якщо воно замінює вибір архітектури та детальне проектування.

## 6.8. Вплив детального проектування на проект

В результаті проведення детального проектування план проекту стає більш конкретним у багатьох відношеннях. Зокрема тепер з більшою точністю може бути проведена оцінка вартості проекту, графіки робіт можуть бути розбиті аж до конкретних завдань, які, у свою чергу, можуть бути розподілені між розробниками.

МОДЕРНІЗУВАННЯ ПРОЕКТУ ПІСЛЯ ЗАВЕРШЕННЯ ДЕТАЛЬНОГО ПРОЕКТУВАННЯ

1. Переконайтеся, що в SDD відображено останню версію детального проектування, зафіксовану після інспектування.

2. Внесіть усі деталі у план-графік (SPMP).

3. Поставте конкретні завдання перед членами команди (SPMP).

4. Уточніть оцінки вартості та тривалості проекту (див. нижче).

5. Виправте SCMP, щоб відобразити нові частини.

6. Здійсніть ревізію процесу, в результаті якого було проведено детальне проектування, та виявіть можливі покращення, включаючи:

♦ витрачений час на підготовку проекту, інспектування та внесення змін;

♦ результати пошуку дефектів. Визначте кількість знайдених, що залишилися, виявлені на стадії деталізації, виправлені при детальному проектуванні дефектів. З'ясуйте, де вони допущені (це стосується і попередніх фаз, і стадій детального проектування).

6.8.1. Оцінка обсягу робіт за допомогою детального проектування

Оскільки за допомогою детального проектування ми можемо оцінити кількість та обсяг необхідних додатку методів, стає можливою і точніша оцінка загального обсягу робіт.

Оцінка об'єму та тривалості проекту за допомогою детального проектування

1. Почніть зі списку методів. Переконайтеся в його повноті, щоб уникнути заниженої оцінки.

2. Оцініть кількість рядків програмного коду для кожного методу.

♦ Класифікуйте методи як дуже малі, малі, середні, великі, дуже великі.

Нормальне співвідношення методів: ±7%/24%/38%/24%/7%.

♦Використовуйте свої дані для переведення оцінок у кількісні значення.

3. Отримайте загальну кількість рядків програмного коду.

4. Перетворіть кількість рядків програмного коду на людину-годинник.

По можливості використовуйте методи перетворення.

Як альтернативу використовуйте опубліковані методи.

5. Переконайтеся, що ваші оцінки розміру методів та тривалості робіт будуть проаналізовані та збережені наприкінці проекту.

## 6.9. Детальне проектування та якість

Якісне виконання детального проектування означає, що ми можемо його зрозуміти. У цьому є й естетичний нюанс; крім того, зрозумілий проект говорить сам за себе. Тим не менш, існують корисні кількісні заходи для оцінки якості проектування. У розділі ми розглянемо кількісні заходи ефективності детального проектування.

Інспектування детального проектування полягає в інспектуванні його класів, прототипів методів цих класів (ім'я, тип, типи параметрів), блок-схем і псевдокода, а також взаємовідносин класів і методів у рамках різних моделей. Ці моделі можуть містити моделі варіантів використання та відповідні їм діаграми послідовності, модель класів, моделі переходів станів та моделі потоків даних.

Інспектування детального проектування:

1. Будьте готові фіксувати метрики під час процесу проектування.

♦ У тому числі час, типи дефектів і строгість вимог.

2. Переконайтеся, що кожний модуль архітектури повністю розкритий.

3. Переконайтеся, що кожна деталь стосується безпосередньо архітектури.

♦ Якщо будь-яка деталь не належить до жодного з модулів архітектури, можливо, потрібно переглянути архітектуру.

4. Переконайтеся, що проектування охоплює всі необхідні функції.

5. Переконайтеся, що проектування є повним (у сенсі класів та методів).

6. Переконайтеся, що проектний варіант тестуємо.

7. Перевірте детальне проектування на:

♦ простоту.

Проектний варіант, який мало хто може зрозуміти (навіть після серйозних зусиль!), досить дорогий для технічної підтримки і може призвести до появи дефектів;

♦ узагальненість.

Чи допускається можливість проектування аналогічних програм;

♦ розширюваність.

Чи допускаються покращення;

♦ ефективність.

Швидкість зберігання даних;

♦ переносимість.

8. Переконайтеся, що забезпечена повна деталізація:

♦ лише програмний код не є «деталлю»

♦ детальна робота все одно має бути колись зроблена, і це найкращий час для її виконання: не відкладайте!

Підсумки:

1. Виконуйте детальне проектування так, щоб воно дозволяло написати програмний код.

2. Намагайтеся використати стандартні зразки проектування.

3. Визначайте вибрані алгоритми:

♦ за допомогою блок-схем;

♦ за допомогою псевдокоду.

4. Використовуйте вибраний інструментарій, наприклад Javadoc.

# 7 Реалізація модулів

Даний конспект є введенням у реалізацію модулів, який розглядається за книгою Еріка Дж. Брауде «Технологія розробки програмного забезпечення».

У цьому вся конспекті розглядаються такі вопросы:

Вибір стандартів.

Вибір стилю кодування.

Кодування з обґрунтуванням коректності.

Визначення вимог якості.

## 7.1 Введення у реалізацію

Написання програм нашвидкуруч дає швидкий, але сумнівний результат; дисциплінований підхід, навпаки, забезпечує більш високу якість із меншими часовими витратами.

У цьому конспекті обговорюються принципи програмування і даються рекомендації, які належать переважно до компаніям, які розробляють програмні програми. Основні положення у контексті розробки програмного забезпечення показано малюнку 2.1.



Рисунок 2.1 – План розробки програм

Приклади:

Оновлення Плану контролю якості програмного забезпечення (SQAP)

Оновлення Плану керування конфігураціями програмного забезпечення (SCMP)

Індивідуальна програмна документація (PSD)

Приклад вихідного коду для класу EncounterCharacter (ПерсонажЗустрічі)

## 7.2 Визначення та мета реалізації

Термін "реалізація" відноситься до програмування. Термін «модуль» відноситься до найдрібніших частин реалізації, які можна підтримувати окремо: це може бути окремий метод чи клас. У прикладах ми вважатимемо методи модулями найнижчого рівня.

Метою реалізації є задоволення вимог у спосіб, визначений у детальному проектуванні. Детального проекту має бути достатньо, оскільки це документ, на якому ґрунтується програмування. Однак часто програміст досліджує паралельно всі попередні документи (архітектуру, D-вимоги та С-вимоги), що допомагає згладити неузгодженість цих документів.

## 7.3 Типова схема процесу реалізації модулів

Типовий процес написання коду програми показано малюнку 2.2.1. Стандарти кодування на малюнку 2.2.1 визначені так, щоб вихідний код мав стандартний вигляд. Архітектура визначає каркас та пакети прикладних програм. Кожен клас кожного пакета потім реалізується через кодування методів, визначених вимогами до детального проектування. Пакети структури необхідно мати на початок розробки програми. Кожен клас перевіряється, як тільки він завершений. Потім кожний клас тестується. Пакет або класи потім випускаються для інтеграції в програму, що розробляється.



Рисунок 2.2.1 – План реалізації модуля

ОДИН З СПОСОБІВ ПІДГОТОВИТИСЯ ДО РЕАЛІЗАЦІЇ

Підтвердіть детальне проектування, яке ви маєте реалізувати. Кодуйте лише записані етапи проектування (частини SDD).

Підготуйтеся виміряти витрачений час, класифікуючи за такими пунктами: залишкове детальне проектування; огляд детального проектування; кодування; огляд кодування; компілювання та виправлення синтаксичних дефектів; тестування модулів та виправлення дефектів, знайдених під час тестування.

Підготуйтеся записати дефекти, упорядкувавши їх за двома характеристиками: серйозність: висока (вимога не виконана), низька або ні те, ні інше; тип: помилка, надання імен, оточення, система, дані, інший.

Прагніть розуміти, які стандарти потрібні: кодування; для персональної документації, яку ви маєте зберігати.

Оцініть обсяг робіт та час на основі ваших попередніх даних.

Плануйте роботу сегментами близько 100 рядків коду.

Деякі з найскладніших методів будуть наведені у псевдокоді чи блок-схемах. Псевдокод із документа детальних вимог може стати коментарем до коду. План тестування розробляється кожного модуля. Кожен із модулів, показаних на діаграмі, тестується. Коли клас реалізований, можна використовувати інструменти зворотного проектування (наприклад Javadoc) для повторної генерації аспектів детального проектування (рисунок 2.2.2).





Рисунок 2.2.2 – Використання зворотного проектування у детальному проектуванні

Зворотне проектування можна порадити після початкової реалізації, оскільки вихідний код часто стає сучаснішим, ніж нижній рівень детального проектування. Це також може бути корисним для успадкованого коду, який погано документований. Взагалі кажучи, зворотне проектування потрібно використовувати тільки коли воно дійсно потрібне. Інженери передчасно "пірнають" у написання коду, а потім підтверджують його розробкою зворотного "проекту", отриманого з коду.

## 7.4 Реалізація в USDP

USDP (Unified Software Development Process – Уніфікований процес розробки програмного забезпечення) дає назви групам ітерацій. Більшість ітерацій слабко впливають на реалізацію, проте ітерації конструювання торкаються основної частини реалізації (рис. 2.3.1).

USDP розглядає реалізацію як ще одну модель (поряд з моделями варіантів використання, тестовою моделлю тощо). Хоча складові проекту і повинні відображатися якомога точніше на відповідні частини фізичної файлової системи, просте відображення може бути непрактично.

Наприклад, кілька класів можна відобразити на один файл і артефакти реалізації можуть містити ознайомлювальні файли README, вихідні файли, об'єктні файли та стислі версії, такі як JAR-файли. Модель реалізації показує, як організовано фізичні артефакти реалізації та як на них відображаються елементи проекту (рис. 2.3.2).



Рисунок 2.3.1 – Реалізація у USDP



Рисунок 2.3.2 – Компоненти моделі реалізації USDP (1)

Модель показує, що класи Зона та ІншийКлас реалізовані в одному файлі - Area.java (ІншийКлас може бути внутрішнім по відношенню до класу Зона).

Модель реалізації складається із вкладених підсистем. Вони складаються з таких компонентів, як файли та інтерфейси реалізації (рис. 2.3.3). Вихідний код для класів Зона та ІншийКлас (рисунок 2.3.3) знаходиться у файлі Area.java. Інтерфейс Зони (набір функцій типу public) у проекті відповідає реально існуючій функціональності, реалізованій в Area.java. Модель реалізації також надає нотацію для відображення зв'язку між файлами, наприклад, "реалізує", "описує" або "стискає" за допомогою лапок (""). (Організація може сама визначати ці зв'язки.)

Наприклад, на малюнку 2.3.3 зазначено, що файл README, вихідний код Зони та його об'єктний код знаходяться в одній колекції і що файл impl.jar – це їх jar-версія (стислий Java-архів).



Рисунок 2.3.3 – Компоненти моделі реалізації USDP (2)

Модель реалізації визначає інтерфейс рівнів класів та мови, який відповідає за реалізацію системного інтерфейсу. Приклад для відеоігри Зустріч із використанням одного інтерфейсного класу показаний на малюнку 2.3.4. За допомогою цієї нотації можна показати кілька класів, які відповідають частині інтерфейсу підсистем. Їх можна об'єднати в інтерфейсний пакет, який відповідає за інтерфейс, і т.д.



Рисунок 2.3.4 – Інтерфейс підсистеми USDP

## 7.5 Мови програмування

Існує безліч мов програмування, від спеціалізованих (наприклад, мови тестування апаратури) до універсальних мов високого рівня, таких як C++, Java та COBOL. Розвиток Інтернету зачіпає широкий спектр мов програмування. Передбачувана мова реалізації впливає проект програмного докладання.

Наприклад, при використанні такої мови, як С, може виявитися краще розробляти проект більш низхідним, функціональним способом, незважаючи на те, що об'єктно-орієнтовані проекти теж можуть певною мірою реалізовуватися на С. Деякі мови, такі як Javascript і раніше Visual Basic, ґрунтуються на концепції об'єктів, що дає програмістам можливість використовувати інкапсуляцію та агрегування, але без наслідування.

## 7.6 Програмування та стиль

Популярне уявлення про програмування як процес підготовки тексту програми для компіляції насправді відбиває лише незначну частину загальної картини. Справжньою метою написання програми є створення правильного коду, тобто такого коду, який повністю відповідає сформульованим для неї вимогам. Компілятори лише перевіряють коректність синтаксису і генерують об'єктний код. За правильність програми відповідає людина, і тому справжній професіонал має переконатися у правильності своєї програми перед тим, як її компілювати. Звичайно, можна і спочатку компілювати, а потім переконуватися в правильності, але це так само неефективно, як спочатку робити синтаксичне виправлення листа, а потім перевіряти, наскільки правильно в ньому викладені думки. Зауважимо також,

Завершує цей розділ рекомендована послідовність кроків написання програм, розроблена з урахуванням матеріалів цього конспекту. Тут виділено типові поради щодо стилю програмування.

ОДИН ІЗ СПОСОБІВ РЕАЛІЗАЦІЇ КОДУ (1)

Сплануйте структуру коду. Заповніть пробіли у детальному проектуванні, якщо такі є. Позначте передумови та постумови. Позначте витрачений час.

Проінспектуйте проект та (або) структуру. Позначте витрачений час, тип, серйозність та джерело дефектів.

Наберіть код: доки не компілюйте; спробуйте використати методи, наведені нижче; застосуйте потрібні стандарти; кодуйте таким способом, який найлегше перевірити (використовуйте формальні методи).

ОДИН ІЗ СПОСОБІВ РЕАЛІЗАЦІЇ КОДУ (2)

Проінспектуйте код, але не компілюйте: переконайтеся, що ваш код виконує необхідну роботу (компілятор ніколи не зробить це за вас: він просто перевірить синтаксис!); позначте витрачений час, тип, серйозність та джерело дефектів; Перегляньте контрольний список інспектування для часто використовуваних деталей конструювання методів та класів.

Відкомпілюйте код: виправте синтаксичні помилки; позначте витрачений час, тип, серйозність та джерело дефектів, число рядків коду.

Протестуйте код.

Загальні засади надійної реалізації

Рекомендується враховувати два загальні принципи у програмуванні:

Спочатку спробуйте повторно використовувати наявний код. Тут розглядається можливість повторного використання існуючого надійного коду, як почати писати новий. Порівняйте, наприклад, використання компонента Java Swing GUI або наявного Java Beans. Насправді, пошук компонентів для повторного використання в Інтернеті зазвичай дає хороші результати.

Принцип "здійснення намірів". Якщо ви розмірковуєте над тим, як ваш код буде використовуватися іншими частинами програми, постарайтеся втілити цей намір у життя. Автор книги називає це принципом "здійснення намірів". Часто з інтерфейсу користувача очевидно, де користувачеві не дозволено вводити заборонені дані. Належить це до внутрішньої обробки.

Якщо ви плануєте використовувати ваш код тільки в окремих випадках, пишіть його так, щоб цей код було неможливо використовувати ніяк інакше.

Принцип «здійснення намірів» аналогічний конструюванню узбіччя та розділових смуг на дорогах для спрямування руху тільки тими шляхами, якими це було заплановано інженерами руху, і ніяким іншим. Такий примус робить дороги безпечнішими і широко використовується в кожній інженерній галузі. Далі наведено приклади принципу "здійснення намірів" у розробці програм.

Використовуйте специфікатори, такі як final, const C++ і abstract, для здійснення відповідних намірів. Класи final не можуть бути успадковані; final-методи не можна переписати у успадкованих класах: значення змінних final не можна змінити. Якщо це призводить до помилок під час компіляції, це означає, що ви не до кінця усвідомлюєте вашу власну програму і жодної шкоди ще не завдали. Особливо ми намагаємось уникнути помилок під час запуску.

Зробіть константи, змінні та класи настільки локальними, наскільки це можливо в межах розумного. Наприклад, визначайте лічильники циклів у циклі: не давайте їм ширшій області використання.

Використовуйте зразок проектування одноелементних множин Singleton, якщо передбачається лише один об'єкт класу.

Загалом робіть члени класу недоступними, якщо немає окремої вимоги забезпечити прямий доступ до них.

Робіть атрибути схованими (private). Доступ до них реалізуйте за потребою через функції загального доступу. (В Java установка типу protected для атрибутів дає об'єктам або підкласу», доступ до членів їхнього базового класу, що часто небажано.)

Робіть методи прихованими, якщо вони повинні використовуватися лише методами того ж класу.

Додайте до документації приклади. Програмісти зазвичай коливаються щодо цього, проте приклади можуть значно допомогти читачеві. У нашому навчальному проекті приклад наведено у коментарях до методу adjustQualityO класу EncounterGame (ІграЗустріч).

Перераховуйте методи за абеткою. Не намагайтеся знайти порядок їхнього виклику. Деякі програмісти люблять групувати методи private, protected та public, а потім розбивають їх на підгрупи статичних та нестатичних методів (у нашому прикладі ця практика не використовується).

Вказівники та посилання

Далі наведено рекомендації.

Уникайте використання змінних-покажчиків у C++ — натомість використовуйте посилання.

Ніколи не повертайте покажчик на нове посилання на купу в C++. Інакше ви будете посилатися на пам'ять, що не використовується, і до обов'язків програміста додасться відновлення простору пам'яті.

Збирайте сміття (C++), використовуючи оператор deleted для всіх непотрібних об'єктів. Невдача при відновленні пам'яті (відплив пам'яті) є основною проблемою в програмах на C++. Деякі організації, що займаються розробкою на C++, дозволяють програмістам виділяти пам'ять лише з допомогою спеціальних службових функцій, щоб поліпшити контроль над процесом виділення пам'яті. Іншими словами, вони намагаються уникати new cd: програмісти повинні використовувати такі утиліти як: F: .-getNewCObjectd для деякого класу F.

Існують також комерційні інструменти, які намагаються визначити потенційно можливі витоку пам'яті. Збір сміття в Java проводиться автоматично, так що витік пам'яті такого масштабу, як у C++, неможливий. Однак Java-програми можуть накопичувати такі ресурси, як файли та сокети, і програміст повинен вести облік ресурсів, що його програма використовує під час виконання.

Функції та виключення

Функції:

Уникайте запитів типів, наприклад if ( myObject instanceOf MyClass ), крім випадків, коли це дійсно необхідно. Натомість використовуйте віртуальні функції.

У C++ уникайте дружніх функцій (за винятком випадків, коли їх переваги явно переважають над недоліками).

Зверніть особливу увагу на перевантаження операцій, оскільки коли вашу програму читатимуть інші люди, вони можуть неправильно зрозуміти зміст ваших операцій. У зв'язку з цим Java не дозволяє навантаження.

Винятки:

Перехоплюйте лише ті винятки, для яких вам відомий спосіб обробки. Метод, що обробляє виняток, має такий вигляд:

ReturnType myMethod( ... ) { ...

try { ... // викликати метод, що генерує ExceptionX } catcht ExceptionX е ) {... // обробити його }

Метод, який не обробляє виняток, виглядає так:

ReturnType myMethod(...) throws ExceptionX{...}

Якщо існуючий метод не може обробити виняток, то повинен бути обробник у зовнішньому коді, який може це здійснити.

Якщо ви можете обробити частину виключення, обробіть її, а потім повторно згенеруйте виняток для обробки у зовнішньому коді.

Робіть розумні припущення щодо можливості операторів, що викликають, обробити ваш виняток або знайдіть альтернативний варіант розробки, оскільки необроблені винятки призводять до аварійних ситуацій у програмі.

Будьте обережні і не використовуйте виняток у ситуаціях, які можуть бути предметом тестування. Наприклад, якщо передумова говорить, що метод ніколи не повинен викликатись з нульовим параметром, це необхідно перевірити тестуванням. Генерація виключення у разі ненульового параметра не повинна бути стандартним способом для обробки такої проблеми.

Якщо вам потрібно вибрати між генерацією виключення і продовженням обчислень, продовжуйте обчислення, якщо можете. Тут на увазі, що програму має сенс закрити тільки в тих випадках, коли ви ретельно продумали наслідки.

Декілька причин, які можуть призвести до неправильної роботи конструктора:

Першою причиною помилки конструктора може бути неправильний аргумент. Рекомендується встановити об'єкт у деякий стан за замовчуванням і лише потім передавати управління обробнику помилок;

друга причина – відсутність необхідних ресурсів. Цю ситуацію найкраще обробити через генерацію виключення, оскільки ми нічого не можемо із цим зробити;

програмісти (особливо на C++) повинні бути особливо уважними під час роботи з пам'яттю, яка буде залишена після генерації винятку у відповідь на помилку конструктора.

Обробка помилок

Розробники постійно стикаються із проблемою обробки некоректних даних. Прикладом некоректних даних може бути номер рахунку, який не відповідає існуючому банківському рахунку. Незважаючи на те, що ми прагнемо робити реалізації якомога простішими, навколишній світ не такий уже простий. Досить більшість процесу програмування пов'язані з обробкою помилок. Тут дуже важливим є дисциплінований підхід: виберіть підхід, сформулюйте його і переконайтеся, що всі інші члени команди розуміють його і будуть його дотримуватися. Один із способів обробки помилок показаний нижче. Він буде пояснений далі.

**ОДИН ІЗ СПОСОБІВ РЕАЛІЗАЦІЇ ОБРОБКИ ПОМИЛОК**

Виконуйте заздалегідь визначений процес розробки; виконуйте інспектування.

Вводячи класи, перевіряйте коректність значень параметрів, що інкапсулюються: private-конструктор, factory-функції для створення об'єктів; при компіляції можна відловити чимало помилок.

Якщо обробку помилок визначено у вимогах, реалізуйте її відповідно до вимог. Використовуйте винятки під час обробки керування під час обробки помилок.

Для програм, які ніколи не повинні відмовляти, намагайтеся передбачити всі можливі дефекти реалізації (наприклад, використовуйте значення за замовчуванням).

Альтернативою може бути дотримання узгодженої політики перевірки параметрів. В основному покладайтеся на якісне проектування та розробку.

## 7.7 Стандарти програмування

Використання стандартів сприяє дисципліни програмування та підвищує читаність та переносимість програм. Ми надамо вашій увазі набір стандартів, які можна використовувати як приклад.

4.1 Угоди про імена: приклади на мові Java

Нижче наводиться приклад угоди про ім'я.

Назвіть об'єкти за допомогою конкатенації слів, наприклад довжина циліндра. Такі імена легко зрозуміти, і вони заощаджують місце. Винятки зі знаком підкреслення залишаються на вибір програміста. Наприклад, замість аАААААААВто було б краще написати: а \_ААА\_АА\_\_А\_ А в т о.

Починайте імена класів із великої літери. Так ви легко відрізнятимете їх від імен змінних. Деякі інструменти починають імена об'єктів стандартними літерами або комбінаціями літер, такими як С... для класів, наприклад, Замовник. Це корисно, коли важливість розуміння типів імен вище за можливу дивність написання.

Імена змінних починайте з маленької літери. Винятком можуть бути константи.

Використовуйте в іменах констант великі літери, наприклад Я\_Є КОНСТАНТУ (використовуйте специфікатори static final). Ім'я ЯЕСТЬКОНСТАНТА було б важко читати; ЯєстьКОНСТАНТУ можна сплутати з класом; їїстьконстанта ніяк не вказує на те, що це дійсно константа.

Починайте (або закінчуйте) імена атрибутів класів підкресленням, наприклад timeOfDay, щоб відрізняти їх від інших змінних, оскільки вони є глобальними по відношенню до всього об'єкта. Ця угода використовується [33], а й критикується [4].

Існує угода, що використовується в нашому прикладі, за яким суфікс I є ознакою атрибута, наприклад timeOfDayl. Кожен атрибут є глобальним для кожного об'єкта класу, і коли таке ім'я зустрічається в блок коду, зручно знати, що це атрибут.

Договоріться про нотацію виділення статичних змінних класу. У прикладі використовується суфікс S, наприклад, numCarsEverBuiltsS. Згадайте, що статична змінна є глобальною для класу, і тому нам корисно знати, що змінна, яка з'явилася у частині коду, є одним із них.

Для методів доступу використовуйте імена, що починаються з get, set та is, наприклад, getNameO, setNameO, isBoxQ (повертає значення Boolean).

**В**як альтернативу можна використовувати пагпеО і name( String ) для імені атрибута (наприклад, CORBA).

Додайте стандартні додаткові методи get і set для колекцій, наприклад insertlntoNameO, removeFromNameO, newNameO.

Умови угоди для параметрів. Можна використовувати префікс а, наприклад, sum(int aNumlP, int aNum2P ), або суфікс P, наприклад sum( int NumlP, int Num2P ).

Документуючи методи, описуйте їх такі характеристики:

передумови та постумови (див. обговорення коректності програми нижче):

що робить метод;

чому він робить те, що робить;

які параметри слід передати (для Javadoc використовуйте тег @param);

винятки, які він генерує (для Javadoc використовуйте тег ^exception);

причина вибору видимості (private тощо);

способи зміни атрибутів;

відомі помилки;

опис тесту із зазначенням, чи тестувався метод, та місце розташування тестового сценарію;

історія змін, якщо ви не використовуєте систему керування конфігураціями;

приклад роботи методу;

спеціальна документація про потокові та синхронізовані методи.

Використовуйте узгоджений стандарт для роздільників. Оскільки одиночні порожні рядки корисні для поділу частин коду всередині методу, узгодженим стандартом можуть бути два порожні рядки між методами.

Договоріться про такі стандарти всередині методів:

виконуйте лише одну операцію на одному рядку;

намагайтеся, щоб ваші методи містилися на один екран;

використовуйте дужки всередині виразів, щоб пояснити їх значення, навіть якщо синтаксис мови цього не вимагає. Це застосування принципу "якщо знаєш це, покажи".

Привласнюючи імена класам, використовуйте слова в однині, наприклад, Замовник, за винятком очевидних випадків, коли ви хочете зібрати об'єкти (у цьому випадку Замовники підійшло б більше). Щоб уникнути розмноження класів, іноді бажано щоб клас збирав всі свої елементи. Це можна зробити за допомогою статичного елемента класу даних.

4.2 Документування атрибутів

Вкажіть всі відповідні інваріанти (обчислювані факти щодо атрибутів, наприклад «36 < Jength \* width < 193''.)

Для кожного атрибуту:

сформулюйте його ціль;

надайте всі відповідні інваріанти (обчислювані факти щодо атрибутів, наприклад «1 < \_аде ​​< 130" або »36 < Jength \* \_width < 193".)

4.3Константи

Перед тим як віднести змінну до типу final (зробити її константою), переконайтеся, що вона має бути такою. Емблер пропонує використовувати не саму константу, а метод, що її представляє.

Наприклад, замість:

protected static final MAX\_CHARS\_IN\_NAME = 20:

слід використовувати метод:

protected final static int getMaxCharsInName()

{

return 20:

}

Цей спосіб виграє завдяки своїй наочності (наприклад, при використанні методу getMaxCharsInNameO замість незрозумілого числа 20) та гнучкості.

4.4Ініціалізація атрибутів

Атрибути завжди повинні бути ініціалізовані, щоб програміст міг контролювати свою програму. Зазвичай ми представляємо ініціалізацію так:

private float balancel = 0: //угода: атрибут xxxl

Однак атрибут, який потребує ініціалізації, може бути об'єктом іншого класу:

private Customer customerl;

Ця ініціалізація часто виконується за допомогою конструктора:

private Customer customerl = new Customer/ Edward, Jones ):

5 Доведено коректні програми

Кажуть, що програма доказово коректна, якщо є математичний доказ того, що вона задовольняє сформульованим стосовно неї вимогам. Доказ ґрунтується на цих вимогах та на коді програми. Воно не пов'язане ні з результатами компіляції, ні з результатами тестування програми. Це ідеальний випадок.

Хороший математичний доказ — це правильно побудована міркування про істинність твердження, що доводиться, переконливе як для самого автора доказу, так і для всіх інших. Ключовим моментом у проведенні доказів є використання мови математики, однак, щоб використовувати цю мову, насамперед потрібно, щоб саме доведення твердження було сформульовано математично.

По-перше, вимога сама собою повинна бути сформульована точно. Тут використовується простіша форма вимог, — передумови та постумови. Передумови визначають усі припущення, зроблені під час виклику функції. Постуслів визначають необхідний стан після завершення виконання функції.

Наприклад, розглянемо таку вимогу до функції f():

Передумова: g – масив цілих чисел довжини п (g «дано»).

Постуслів'я: г = max{g[0], g[l], ..., g[n-1]} (г - "результат").

У неформальних термінах ця вимога викликає функцію f() для визначення максимального значення масиву g. Це легко запрограмувати, однак ми зробимо так, щоб нам було легко довести коректність коду.

Одним із способів познайомитися з програмуванням фрагментів коду, коректність яких можна довести, є застосування (псевдокоду):

*while (Змінним присвоєно невирішені значення)*

*Виконати відповідні дії*

*/\*0сталося лише довести, що цей цикл кінцевий, щоб переконатися, що змінним будуть надано необхідні значення\*/*

Для підтвердження того, що змінній надається необхідне значення, програмісту достатньо переконатися в кінцівці циклу. Якщо програміст може довести, що цикл скінчен, твердження «Змінної не присвоюється необхідне значення» не буде справедливим (подвійне заперечення), і необхідний стан буде досягнуто.

На перший погляд, коректні програми, що доводяться, здаються новим напрямком у програмуванні. Це не зовсім так. Інтуїтивний процес, який зазвичай використовується програмістом, по суті нічим не відрізняється від формального доказу: формальність просто допомагає контролювати процес із набагато вищою точністю. Наприклад, більшість із нас програмуватимуть функцію тах() за допомогою створення циклу та накопичення максимального значення «з розглянутих».

Іншими словами, під час компіляції ми зберігатимемо наступне твердження як істинне:

[/' <= п-1] AND [г = maximum} g[0], g[l], ..., g[j] }] //затвердження I

6 Інструменти та середовища програмування

Часто кажуть, що людина сама створює інструменти, і це також стосується розробників програм. Існує кількість інструментів, що допомагають у програмуванні.

Інтерактивні середовища розробки широко використовуються, допомагаючи програмістам виробляти більше коду за менші терміни. Сюди відносяться можливість перетягування для конструювання компонентів графічного інтерфейсу користувача, графічне уявлення каталогів, відладники, майстри і т. д. У Java Beans використовується підхід, що стандартує вихідний текст таким чином, що середовище розробки Java Beans може обробляти вихідні дані. Перевага цього підходу полягає в тому, що розробники не прив'язані до одного середовища розробки. Значно спростилося створення СОМ-об'єктів.

*Програми профілювання,*такі як Jprobe, можуть використовуватися для накопичення статистичних даних, наприклад:

сукупний час центрального процесора і час, що минув;

час, витрачений кожен метод;

сукупна кількість створених об'єктів;

кількість викликів;

Середній час роботи кожного методу.

7 Якість у реалізації

У цьому розділі обговорюються метрики та методи досягнення якісних реалізацій. Нижче наведено контрольні списки інспектування коду.

ОДИН ІЗ СПОСОБІВ ІНСПЕКТУВАННЯ КОДУ (1): КЛАСИ В ЦІЛОМУ

Чи підходить ім'я для класу: а) Узгоджується з вимогами чи проектом? б) Чи достатньо конкретно (узагальнено)?

Чи може бути клас абстрактним (щоб використовуватися тільки як основа)?

Чи визначає заголовок класу його ціль?

Чи заголовок класу посилається на вимоги елемента проекту, якому цей клас відповідає?

Чи вказано там пакет, до якого цей клас належить?

Чи є клас настільки прихованим, наскільки це можливо?

Чи не зробити його final (Java)?

Чи використовувалися стандарти документації? Наприклад, Javadoc.

ОДИН ІЗ СПОСОБІВ ІНСПЕКТУВАННЯ КОДУ (2): АТРИБУТИ

Чи є атрибут справді необхідним?

Чи можна зробити його статичним? Чи потрібна у кожному класі своя змінна?

Чи не зробити його final? Чи дійсно змінюється його значення? Чи достатньо одного методу get?

Чи правильно було застосовано угоди про присвоєння імен?

Чи є атрибут настільки прихованим, наскільки це можливо?

Чи є атрибути настільки незалежними, наскільки це можливо?

Чи існує ясна стратегія ініціалізації: а) на момент оголошення? б) із конструктором (конструкторами)? в) використовуючи Statiс {}? г) все вищезгадане? Як?

7.1 Стандартні метрики для вихідного коду

*Число рядків коду*хоч і не є ідеальним показником, але є однією з важливих програмних метрик. Необхідно вибрати стандартний спосіб підрахунку рядків.

Наприклад, потрібно вирішити, як враховувати за підрахунком:

твердження, що займають кілька рядків (1 або й?);

коментарі (0?);

рядки, що складаються з while, for, do тощо (1?).

Вибір може залежати від застосованого стандарту і від інструментарію, що здійснює автоматичний підрахунок. Дотримання обраного способу підрахунку важливіше, ніж сам вибір цього способу.

**IEEE-метрика 14. Програмні наукові метрики**

**Вимірювані величини:**

Нехай і — це кількість окремих операторів (+, \* і т. д.) у програмі. Наприклад, у програмі (х = х + у; г = х \* w} щ = 2).

Нехай п2 = число окремих операндів у програмі. Наприклад, у програмі {х = х + у; z = х \* w} п2 = 4, оскільки оператори складання та множення мають по два операнди кожен.

Нехай JV = число всіх операторів у програмі.

Нехай N2 = число всіх операндів у програмі.

**Приклад оцінки:**

передбачувана довжина програми: n, (log п {) + n2 (\ og п2);

складність програми: (nlN2)/(2п2).

7.2 Індивідуальні метрики для вихідного коду

Цикломатична складність виражає число циклів у програмі, але не проводить різницю між вкладеними та неукладеними циклами. У загальному випадку цикли всередині циклів схильні до помилок частіше, ніж послідовності незалежних циклів.

Неважко створити метрики, які вимірюють важливі величини. Наприклад, щоб обчислити складність циклу, можна припустити, кожен вкладений цикл збільшує складність на порядок (грубо кажучи, в 10 разів).

Наведений нижче приклад можна розцінювати як опис коефіцієнта складності вкладених циклів.

Цикл 1 – складність 1.

Цикл 2 – складність 1.

Цикл 2.1 – складність 10 (цикл усередині циклу 2).

Цикл 2.2 – складність 10.

Цикл 2.2.1 – складність 100.

Цикл 3 – складність 1.

Цикл 4 – складність 1.

Цикл 5 – складність 1.

Цикл 5.1 – складність 10.

Використовуючи цю метрику, отримуємо складність 135. Цю метрику можна нормувати, розділивши отриману величину число рядків коду і порівнявши результат із середнім значенням по фірмі.

 7.3Інспектування коду

 Згадайте, що серйозність помилок є важливою складовою інспектування даних, оскільки вона дозволяє розставляти пріоритети дефектів і цим раціонально розподіляти роботу (таблиця 7.3.1). Нема рації витрачати час інспектування на тривіальні дефекти. Непоганим способом обробки тривіальних дефектів є ситуація, коли інспектори передають роздруківку вихідного коду із зауваженнями автору.

Таблиця 7.3.1 – Класифікація серйозності дефектів з використанням методу відбраковування



Крім того, дефекти можна описати, ґрунтуючись на їхньому типі; одна з таких класифікацій:

Проблема логіки (забуті випадки або кроки; повторена логіка; пропуск граничних умов; неважливі функції; некоректне переривання; брак тексту умови; перевірка неправильних змінних; некоректність ітераційних циклів тощо).

Обчислення (некоректне чи недостатнє рівняння; втрата точності; помилка у знаку).

Інтерфейс та синхронізація (некоректна обробка переривань; некоректна синхронізація вводу-виводу, невідповідність модулів або підпрограм).

Обробка даних (некоректна ініціалізація даних; некоректне отримання або зберігання даних; некоректний масштаб або одиниці виміру даних)

Кордони даних.

Дані (некоректні або відсутні дані датчиків, дані оператора, дані таблиць, зовнішні дані, вихідні дані, вхідні дані).

Документація (неоднозначний опис тощо).

Якість документа (невідповідність застосовуваним стандартам тощо).

Розширення (зміна у вимогах тощо).

Помилки, що виникли внаслідок попередніх виправлень.

Можливість взаємодії (несумісність з іншими програмами чи компонентами).

Відповідність стандартам.

7.4 Індивідуальна програмна документація (PSD)

Кожен розробник зобов'язаний зберігати документацію своєї роботи. Ця документація називається індивідуальною програмною документацією (PSD - Personal Software Documentation). Вона дозволяє в будь-який момент отримати інформацію про статус розробки, і частину її пізніше буде додано до архіву проекту. Ця документація може містити вихідний код та дані про індивідуальну реєстрацію дефектів, що складаються з наступних пунктів.

Типи дефектів.

Індивідуальний етап, у якому дефект виник.

Індивідуальний етап, де дефект був виправлений.

Індивідуальні етапи - це:

додаткове детальне проектування (якщо застосовується);

код (записуйте виниклі або виявлені дефекти у вихідному коді до компілювання);

звіт про компіляцію (записуйте виявлені та виправлені дефекти після спроби компіляції);

модульне тестування Модульне тестування проводиться групою контролю якості та не є частиною цієї документації.

Облік часу. Час, витрачений на додаткове детальне проектування, кодування, компіляцію та тестування.

Нотатник проектування. Містить статус додаткового детального проектування (якщо застосовується) та коду, а також відомості про інциденти та питання розробки, що потребують уваги.

8Підсумки процесу реалізації

Метою реалізації є коректне програмування детального проекту за підтримки експлуатаційної надійності продукту. Використання стандартів програмування допомагає у цьому процесі.

Дуже важливо мати організований підхід щодо мети кожного методу, і цього можна досягти шляхом формулювання передумов і постумов. Щоб коректність коду, особливо потенційно складного коду, можна було довести, необхідно писати текст програми організованим способом, що бажано припускає формальний доказ. Це полегшує підтвердження задоволення вимогам. Підведемо підсумки.

Завжди пам'ятайте про мету програмування: коректність; чіткість.

Використовуйте стандарти програмування.

Визначте передумови та постумови.

Доведіть правильність окремих частин без компілятора.

Реалізуйте програму за заданий час.

Підтримуйте якість та професіоналізм.

# 8 Модульне тестування

У зв'язку з потенційною шкодою від кожної програмної помилки та зростаю­Ще складністю знаходження та виправлення помилок по ходу зростання програми раннє і часте тестування стає важливою частиною процесу розробки. Відразу після реалізації елементів програми ми негайно приступаємо до тестування.

Розділи 8.1-8.6

Вправи

Приклад: Індивідуальна програмна документація (PSD), частина 2 Питання, що розглядаються в цьому розділі:

Розуміння значення модульного тестування.

Відмінність між тестуванням методами «чорного» та «білого ящика».

Розробка адекватної зони охоплення.

Вивчення стандартів тестування.

Інспектування плану модульного тестування

## 8.1. Введення у модульне тестування

8.1.1. Цілі тестування

Ми не можемо протестувати програму абсолютно у всіх аспектах, оскільки кількість варіантів роботи нетривіальної комп'ютерної програми може бути необмеженою. Отже, тестування неспроможна довести відсутності помилок у програмі, тоді як доказ коректності здатне це. Тестування може лише показати наявність помилок.

Тестування часто неправильно сприймається як процес підтвердження коректності коду, що можна висловити таким висловом: "Протестуй це, щоб переконатися, що тут все правильно". Іноді це є метою тестування, особливо незадовго до відсилання продукту або при регресійному тестуванні (пояснюється в наступному розділі). Головна мета тестування далека від підтвердження коректності. Його мета не в тому, щоб показати задовільну роботу програми, а в тому щоб чітко визначити, в чому робота програми незадовільна!



8.1. План розробки програм

Час, використаний для тестування, вимагає значних витрат, і ми намагаємося отримати від цих витрат максимальний прибуток. Для даної програми, що тестується, чим більше дефектів буде знайдено на кожен долар зарплати, тим вище виграш від вкладень у тестування. Отже, метою тестування є виявлення якнайбільшого числа дефектів з високим рівнем важливості. Резюмуючи сказане вище, перерахуємо "золоті правила" тестування.

♦ Мета тестування: максимізувати число та важливість виявлених дефектів на кожний витрачений долар.

♦ Тому: починайте тестування зарано.

♦ Обмежені можливості тестування: тестування може встановити лише присутність дефектів, і ніколи їх відсутність.

♦ Для встановлення факту відсутності дефектів використовуйте докази коректності

Тестування оцінюється більш ніж половиною часу, витраченого на проект. Нагородою за знаходження дефекту на ранній стадії процесу є принаймні десятикратна економія в порівнянні з виявленням цього дефекту на етапі інтеграції або, ще гірше, після відправки замовнику. Отже, ми маємо тестувати рано та часто.

Щодо ідеальної гарантії якості загалом, тестування коду повинні проводити люди, які не брали участь у його розробці. Коли інженер розробляє код, він створює собі уявлення те, що код повинен виконувати. Тому в той же час він розробляє типове середовище, в якому цей код має виконуватись. Можна сміливо вважати, що код дає трохи помилок у цьому конкретному середовищі. Отже, це середовище є основою тестів розробника. Саме тому коли людина тестує свій власний код, вона часто ховає кожен дефект, який необхідно знайти.

p align="justify"> Модульне тестування є раннім типом тестування. Наступний рівень складається із інтегрального тестування. Тут валідується загальна функціональність кожної стадії конкретної програми. Нарешті, система та різні приймальні тести валідують фінальний продукт, як описано в наступному розділі. Вже розроблені варіанти використання також беруться як основа для деяких із цих тестів. Типи тестування та зв'язок між­ними проілюстровані на рис. 8.2



8.1.2. Значення модульного тестування

Мета модульного тестування - перевірити структуру, у той час як мета всіх інших видів тестування зазвичай полягає у перевірці функціональності. Як аналогія уявіть собі тестування кожної опори мосту на заводі. Це є подібністю модульного тестування, оскільки в цьому випадку тест зачіпає елементи структури. Тест, що складається з проїзду автомобіля частково сконструйованим мостом, навпаки, не буде модульним тестуванням. Функції зазвичай є найменшими частинами програми, яких може бути застосовано модульне тестування (див. рис. 8.2). Наступним за величиною елементом є модуль (клас у разі об'єктно-орієнтованої орієнтації). Іноді комбінації модулів розглядаються з метою тестування як модулі.

Модулі, до яких застосовується модульне тестування, є блоками при побудові програми, а не окремою цеглою, на якій будується будинок. Хоча на будинок не сильно вплинуть кілька бракованих цегли, програмне застосування може виявитися дуже чутливим до дефектів в окремих блоках конструкції. Якщо дефектні частини будуть вбудовані в програми, може знадобитися багато часу на їх знаходження та виправлення. Тому блоки програми повинні бути абсолютно надійними, що є метою модульного тестування.

8.1.3. Типовий план модульного тестування

Типовий план модульного тестування, що базується на стандарті IEEE 1008. 1987, показаний на рис. 8.4. Далі пояснюються кроки процесу модульного тестування.

Вхідними даними для процесу планування тесту є вимоги та детальний проект. Згадайте, що кожна вимога відповідає добре визначеному коду (функції, де можливо). Детальний проект зазвичай складається з додаткових класів та методів. Вони також позначаються на якості програми і мають бути протестовані в тому ж обсязі, що й окремі вимоги. Вихідними даними процесу планування тесту є модульний план тестування (наприклад, "(1) тест методу 84; (2) тест методу 14; ...; (m) тест класу 26, ...").

Наступним кроком є ​​отримання вхідних та вихідних даних, що асоціюються з кожним тестом. Деякі з цих даних можуть бути отримані з попередніх тестів (наприклад, попередні ітерації, попередні версії продукту або попередні випуски). Результатом цього кроку є набір тестів

Нарешті, тести виконуються



Далі наведені етапи, необхідні стандартом IEEE для модульного тестування (1008-1987). Вони розширюють щойно описаний план модульного тестування

1. Спланувати загальний підхід, ресурси та розклад.

2. Визначити характеристики, які слід протестувати, з вимог.

3. Відновити загальний план.

4. Розробити набір тестів.

5. Реалізувати оновлений план та проект.

6. Виконати тестові процедури.

7. Перевірити завершення роботи.

8.2. Типи тестів

8.2.1. Тестування «чорної скриньки», «білої скриньки» та «сірої скриньки»

У цьому розділі буде визначено тестування «чорного», «білого» та «сірого ящика». В решті розділу буде описано, як планувати, проектувати та виконувати такі тести. Коли ми цікавимося виключно тим, як програма або її частина надає відповідні вихідні дані, ми тестуємо її на кожну вимогу, використовуючи відповідні вхідні дані. Це називається тестуванням «чорної скриньки», оскільки ми не звертаємо уваги на те, що знаходиться всередині «скриньки» (програми): «скринька» може бути «чорною». Тести «чорної скриньки» можуть бути ефективними, якщо ми можемо переконатися, що вони вичерпують усі комбінації вхідних даних. Це доведе замовнику, що всі вимоги задоволені. Проте, жодне тестування не охоплює всіх можливостей. Тестування «чорної скриньки» схоже на тестування мосту шляхом проїзду кількома комбінаціями різних транспортних засобів. Це неефективно, оскільки нам потрібно перевірити і складові мосту, і те, як вони об'єднані в систему. Останнє називається ідеєю «тестування білої скриньки». Тестування «чорної скриньки» та «білої скриньки» проілюстровано на рис. 8.5.



Метою тестування «білої скриньки» є тестування найбільш ненадійних шляхів програми. Для виконання тестування «білої скриньки» ми спочатку розбиваємо проект програми на окремі елементи та шукаємо шляхи та інші розбиття для керування та даних. Потім ми проектуємо тести, що простежують усі або деякі з цих шляхів, і перевіряємо всі складові. Наочнішою назвою цих дій було б «тестування скляного ящика».

Тестування «сірої скриньки» розглядає внутрішню роботу програми чи модуля, але лише певною мірою. Сюди можуть бути також віднесені деякі аспекти тестування «чорної скриньки».

8.2.2. Розбиття рівнозначності для тестування «чорної скриньки»

Оскільки ми не маємо можливості протестувати всі комбінації вхідних даних, ми шукаємо представницькі варіанти тестів. Набір можливих варіантів тестів для трьох змінних у фінансовій програмі – капітал, процентна ставка та оцінка інфляції – зображено на рис. 8.6. Проблема полягає у знаходженні найкращого уявлення нескінченної множини можливостей найбільш представницьким певним множиною. Розбиття рівнозначності - це розбиття вхідних тестових даних на підмножини таким чином, щоб за умови успішного виконання одного з них інші елементи підмножини також з великою ймовірністю пройшли б тест успішно



Розбиття рівнозначності для методу, що обчислює величину в залежності від заданого капіталу, процентної ставки та оцінки інфляції, представлено на рис. 8.7. Наприклад, область зафарбована на рис. 8.7 сірим, відповідає наступному розбиття рівнозначності: "оцінка інфляції між 15 і 20%", "капітал від $65 до $100" та "відсоткова ставка між 0 і 5%". Максимальний прибуток від тестування зазвичай досягається при аналізі граничних значень, про який йтиметься далі.



8.2.3. Аналіз граничних значень для тестування «чорної скриньки»

До розбиття рівнозначності зазвичай вдаються для дослідження граничних значень внутрішніх змінних програми. Наприклад, оцінка інфляції має лежати між 1 та 20 %, що дає дві межі. Припустимо, значення інфляції до 15 % і значення, перевищують цю величину, програма обробляє по-різному. Це дає ще один додатковий кордон (рис. 8.7).

У тестуванні проекту значення, що лежать за межами цих кордонів (наприклад, недійсне введення даних), також використовуються як тестові дані. Тестові дані генеруються після встановлення межі еквівалентних класів (рис. 8.8).



8.2.4. Розгляд тверджень для тестування «білої скриньки»

Кожне затвердження у програмі має бути виконане принаймні одним тестом. Аналіз кожного твердження є обов'язковим. Як приклад (рис. 8.9) покажемо, що виконання всіх тверджень не гарантує коректності програми (див. [82]). Вимоги до програми на рис. 8.9 визначено блок-схемою. Тестовий варіант u=2, v=0, х=3 виконує кожну команду реалізації та генерує коректні вихідні дані (х=2,5). Однак програма дефектна, тому що в ній не реалізовано блок-схему. Наприклад, у разі u=3, v=0. х=3 програма генерує відповідь х=1, хоча має бути х=2, як показано на блок-схемі.



8.2.5. Розгляд рішень для тестування «білої скриньки»

Огляд рішень гарантує, що програма приймає кожну гілку кожного рішення. Наприклад, розглянемо блок-схему на рис. 8.10. Переконаємося, що кожне так і ні можна отримати в результаті хоча б одного тесту з набору тестів. Цикли ефективно реалізують послідовність умовних операторів. Наприклад, цикл

for( 1 = 0; I < 3; ++i)

v[i] = w[I+l] + w[i];

можна «розкрити» в таку послідовність умовних операторів:

// For i == 0:

i = 0;

v[i] = w[i+l] + w[i]:

++i;

// For i ==1:

if (i < 3)

{ v[i] = w[i+l] + w[i]:

++i : }

// For i==2:

if (i < 3) {

v[i] = w[i+l] + w[i]: + + i; }



Огляд рішень забезпечує виконання кожної ітерації кожного циклу. Його можна реалізувати шляхом визначення кожного циклу та підбору вхідних даних таким чином, щоб кожна ітерація кожного циклу виконалася хоча б один раз. Цей підхід не такий складний для циклів for, однак він представляє деякі складнощі для циклів while.

Іноді можливі варіанти можна прорахувати, іноді їх можна розбити на типові групи. Однак у деяких випадках повне розгляд рішень з допомогою циклів while практично неможливо. Згадайте, що цикли while часто допускають застосування формальних методів та інспектування

8.2.7. Використання випадкових величин у тестуванні

Взагалі кажучи, ми використовуємо випадкові вхідні дані при тестуванні з метою уникнення необ'єктивних екземплярів тестів. Наприклад, якщо ми хочемо оцінити думку населення про політичного діяча, ми обираємо представників населення випадковим чином. Тут випадковий вибір застосовується певним чином: спочатку визначається тип тесту, а потім застосовується довільна вибірка для отримання об'єктивних тестових даних. Те саме справедливо і для програмного тестування. Коли вибрано тип тесту (наприклад, розгляд рішень) та визначено межі даних, для можливих вхідних даних існує певна свобода. В ідеалі ці вхідні дані мають вибиратися випадковим чином. Як приклад уявіть виконання розгляду рішень для блок-схеми з розділу 8.2.4. Припустимо, ми хочемо, щоб відповіддю на запитання «Ім'я надто довге?» було «Так». Існує безліч даних, для яких це буде справедливо, і саме з цієї множини і слід в ідеалі випадковим чином вибирати тестовий приклад. Один із способів – вибрати кожне ім'я, як зазначено далі.

Вибрати цілісне i більше, ніж maxNumCharsInName() випадковим чином. Для; - 0...г вибрати випадковим чином букву та встановити в name[j] цю букву. Як другий приклад представте виконання тестування граничних значень для програми, що перевіряє, чи є чотири речові числа з послідовності х1, х2, хЗ і х4 допустимими значеннями для експерименту.

8.3. Планування модульних тестів

Систематичний підхід у тестуванні необхідний, оскільки кількість потенційних модулів, що потребують тестування, зазвичай дуже велика. Досить легко сказати, що «кожна частина роботи має бути протестована», проте ця фраза несе у собі мало сенсу, оскільки на етап тестування виділяється лише обмежена кількість ресурсів. Тому мета полягає у знаходженні якомога більшої кількості помилок якомога серйознішого рівня в рамках наявних ресурсів

ОДИН З СПОСОБІВ СПЛАНУВАТИ МОДУЛЬНЕ ТЕСТУВАННЯ

Визначте принципи модульного тестування.

♦ Призначити самого розробника відповідальним?

♦ Надати перевірку іншим?

♦ Надати проектування та перевірку іншим?

Визначте, що, де та як документувати.

♦ Індивідуальна документація?

♦ Як і де впроваджувати до інших типів тестування?

♦ Чи впровадити у формальні документи?

♦ Чи використовувати інструменти та тестові утиліти?

Визначте обсяги модульного тестування заздалегідь:

♦ не тестуйте просто «поки час не скінчиться»;

♦ розставляйте пріоритети, щоб важливі тести були точно проведені.

4. Визначте, як і де отримати тестові дані.

5. Оцініть потрібні ресурси.

♦ Використовуйте наявні дані попередніх проектів, якщо це можливо.

6. Організуйте облік часу, підрахунок помилок, фіксацію їх типу та джерела

1. Визначте принципи модульного випробування. Перше питання полягає у визначенні того, які модулі ми розглядатимемо, і хто їх тестуватиме. Для об'єктно-орієнтованих проектів звичайна організація модульного тестування полягає у тестуванні методів кожного класу, потім класів кожного пакета, потім пакета загалом. У нашому прикладі ми спочатку тестуватимемо класи в кожному пакеті, а потім перейдемо до програмних пакетів, оскільки тестування останніх залежить від тестування перших. Тестування модуля в ідеалі планується і виконується людиною, яка не брала участі в розробці. Модульне тестування у реальному житті іноді планується та виконується організацією контролю якості. Хоча гідністю такого підходу є незалежність тестування, у цьому випадку від інженерів організації контролю якості потрібне розуміння проекту в деталях. Деякі організації-розробники не підтримують цю можливість і тому вимагають від контролюючої якості організації лише тестування високого рівня. Модульне тестування часто залишається за групою розробників і виконується на власний розсуд. У будь-якому випадку тести стають доступними для інспектування і можливого застосування вищого рівня. У певному сенсі, незалежність організації контролю якості може бути замінена перехресним тестуванням, що виконується самими розробниками, коли вони тестують модулі один одного. Модульне тестування часто залишається за групою розробників і виконується на власний розсуд. У будь-якому випадку тести стають доступними для інспектування і можливого застосування вищого рівня. У певному сенсі, незалежність організації контролю якості може бути замінена перехресним тестуванням, що виконується самими розробниками, коли вони тестують модулі один одного. Модульне тестування часто залишається за групою розробників і виконується на власний розсуд. У будь-якому випадку тести стають доступними для інспектування і можливого застосування вищого рівня. У певному сенсі, незалежність організації контролю якості може бути замінена перехресним тестуванням, що виконується самими розробниками, коли вони тестують модулі один одного.

2. Визначити метод документування модульних тестів. Документування модульних тестів складається з тестових процедур, вхідних даних, коду, що виконує тест, та вихідних даних. Модульні тести можна упаковувати разом із кодом чи окремих документах. Гідність упаковки тестів разом із кодом полягає у зручності. Цю техніку проілюстровано у розділі 8.4.2.1. Недоліком є ​​збільшення обсягу коду. Можна використати попередню компіляцію для урізання тестового коду перед компіляцією модуля. Крім того, для виконання модульних тестів використовуються драйвери тестів та утиліти. Вони також документуються для майбутнього використання

3. Визначте обсяги модульного тестування. Оскільки «протестувати все» неможливо, межі тестування мають бути свідомо визначені. Наприклад, якщо банківська програма складається зі зняття, депозитів та запитів, модульне тестування могло б вказати, що кожен метод повинен бути протестований з рівною кількістю дозволених, граничних та заборонених даних або, можливо, методи зняття та депозиту мають тестуватися втричі більше, ніж методи запитів, тощо. буд. У випадку методи, змінюють стан (значення змінних), зазвичай тестуються більше за інших. Межі того, що відноситься до модульного тестування, також мають бути визначені. Наприклад, чи сюди входить тестування пакетів, чи воно має відноситися до іншого типу тестування (глава 9). Розробники заздалегідь визначають межі тестування, у тому числі й момент, коли процес тестування має бути завершено. Наприклад, чи слід тестувати кожен модуль однакової кількості часу або до виявлення перших трьох помилок? Критерії зупинки тестування обговорюються в розділі 8.5.2.

4. Визначте, як і де отримувати вхідні тестові дані. Ми обговорили дозволені, граничні та заборонені вхідні тестові дані. Також потрібна деяка випадкова генерація даних. По можливості використовуються інструменти, що генерують вхідні тестові дані за допомогою аналізу вихідного коду та виявлення граничних значень даних та розгалуження. До того ж значний обсяг тестових даних можна отримати з попередніх версій програми, стандартних джерел, промислових контрольних завдань тощо. Усе це документується для майбутніх посилань та повторного використання.

5. Оцініть потрібні ресурси. Як завжди при плануванні, ми визначаємо людино-місяці та час, необхідний для виконання модульного тестування. Основним джерелом цієї оцінки є накопичені дані. Хоча модульне тестування часто пов'язане з процесом розробки, його окреме виконання надає безцінну інформацію.

6. Організуйте облік часу, а також облік дефектів, їх типу та джерела. Інженери, що беруть участь, визначають чітку форму, в якій вони будуть вести облік витраченого на модульне тестування часу, облік помилок та їх типів. Отримані дані використовуються для затвердження стану програми та передбачення кінцевої якості роботи та термінів закінчення. Дані також стають частиною облікових записів історії проекту. Наступні два розділи представляють приклади планування модульних тестів на рівнях методів та класів. Приклад в кінці розділу показує код, що вийшов.

8.5.4. Тести на основі станів

Як ми бачили, об'єкти класів часто можна як перехід між станами у відповідь події. Такі класи ми повинні протестувати на основі їхнього стану. Наприклад, протестуємо клас ГраЗустріч. Перші кроки у тестуванні станів програми показано на рис. 8.12. Повний тест переходів станів показано на рис. 8.13. Числа показують типовий порядок подій, що призводить до послідовності подій, що має сенс. Один тест тому повинен бути подачею таких вхідних даних, щоб система проходила цю послідовність станів. Також слід розглянути події, які не застосовні до жодного конкретного стану, щоб переконатися, що вони не впливають на хід роботи програми





На планування та виконання тестів на основі станів необхідно витратити значний час. Це в першу чергу пов'язано з необхідністю вичерпних вхідних даних. Існують інструменти, що записують взаємодію користувача з системою, які можуть відтворити ці дії. Крім цього, можна використовувати перевірку інваріантів у коді, щоб переконатися в тому, що система знаходиться в тому стані, в якому повинна.

8.6. Підбиття підсумків

Модульне тестування складається з тестування структурних компонентів програми, що знаходиться в розробці. Цим вони відрізняються від комплексних тестів, що перевіряють функціональні частини програми в процесі розробки. Тестування «чорної скриньки» перевіряє, щоб код, що тестується, надавав коректні вихідні дані за всіма вхідними даними. Тестування «білої скриньки» перевіряє процес отримання вихідних даних. У тестування «білої скриньки» є кілька аспектів. Тестувати слід рано та часто. Розроблені плани тестів заздалегідь допомагають пролити світло на вимоги. Підіб'ємо підсумки цього розділу.

♦ Модульне тестування – тестування «частин».

♦ Інтегральне тестування – тестування «всього в цілому».

♦ Метод «чорної скриньки» перевіряє обробку:

- Виконувану багатьма способами;

- дозволяє переконатися у закінченості.

♦ Планування тесту: що раніше, то краще.

♦ Допомагає прояснити вимоги.