Лекція 10.
Культура програмування

Зміст

Цільова настанова: Розглянуто проблематику, цілі та вимоги до курсу, наведена мотивація до вивчення технологій тестування програмних продуктів, проаналізовано історію розвитку тестування програмного забезпечення, введено термін і розглянуто поняття якості програмного продукту, розглянуто якість програмного забезпечення в контексті міжнародних стандартів проведено аналіз моделі якості програмного продукту, розглянуті питання створення, використання і аналізу метрик якості програмних продуктів..

Ключові слова: тестування, якість програмного продукту, мотивація, міжнародні стандарти, моделі якості програмного продукту, метрики якості програмних продуктів.

1. Якість програмного продукту

Кожен день у своїй роботі ми стикаємося з досить абстрактним поняттям «якість ПО» і якщо поставити запитання тестувальників або програмісту - «що таке якість?», То у кожного знайдеться своє тлумачення.

Якість програмного продукту характеризується набором властивостей,

що визначають, наскільки продукт «хороший» з точки зору зацікавлених сторін, таких як замовник продукту, спонсор, кінцевий користувач, розробники і тестувальники продукту, інженери підтримки, співробітники відділів маркетингу, навчання і продажів. Кожен з учасників може мати різне уявлення про продукт і про те, наскільки він хороший чи поганий, тобто про те, наскільки висока якість продукту.

Таким чином, постановка задачі забезпечення якості програмного продукту виливається в задачу визначення зацікавлених осіб, їх критеріїв якості і потім - знаходження оптимального рішення, що задовольняє цим критеріям.

Отже, що ж таке якість програмного забезпечення?

Спробуємо дати визначення термінам якість і якість програмного забезпечення. Основною проблемою є той факт, що визначення якості занадто неясне і неоднозначне. Це викликано тим, що зазвичай термін якість розуміється неправильно. Така плутанина може пояснюватися кількома причинами.

• Перша, якість - це не окремо взята ідея або поняття, швидше за багатовимірна і неоднозначна концепція.

• Друга, для будь-якого поняття і визначення існує кілька рівнів абстракції, наприклад, коли люди говорять про якість, одна частина розуміє під цим занадто широкий і розмитий сенс, в той час як інша може посилатися на конкретне визначення і значення.

• Третя, термін якість є невід'ємною частиною нашого повсякденного спілкування, проте загальноприйняте і професійне використання може бути досить сильно відрізнятися.

Популярний погляд на якість. Загальноприйнята думка про якість таке, що це щось нематеріальне і "невловиме" - про нього можуть вестися суперечки і дискусії, його можна критикувати і вихваляти, але зважити і виміряти якість неможливо. Такі вирази як «хорошу якість» і «погана якість» служать наочним прикладом, як люди кажуть про щось невизначеному для них, то, що вони не можуть чітко характеризувати і визначити. Таку думку відображає той факт, що люди сприймають і інтерпретують якість по-різному.

Мається на увазі, що якість не може бути контрольованим і керованим, і тим більше воно не може бути кількісно вимірюваних. Інша популярна думка, що якість нерозривно пов'язане з розкішшю, першим сортом і тонким смаком. Дорогий, досконально продуманий і більш технічно складний продукт розглядається як гарантія високої якості, ніж дешевші аналоги.

Відповідно до такого підходу, якість обмежено певним класом дорогих продуктів з хитромудрої функціональністю і класовим продуктам. Простіше кажучи, чи недорогий продукт буде класифікований як якісний продукт. На жаль, таке невизначене і розпливчасте уявлення не може бути використано для поліпшення процесів розробки програмного забезпечення і явно контрастує з професійним підходом до управління якістю.

Професійний підхід до управління якістю стверджує, що якість - це чітко визначена величина, яку можна виміряти і проконтролювати, вона піддається

управлінню і поліпшенню. Спробуємо дати чітке і зручне для роботи визначення. У літературі в 1970 році якість було визначено як «відповідність вимогам», а в 1979 його визначили, як «придатність до використання». Ці два визначення тісно пов'язані і прекрасно узгоджуються, як ми побачимо пізніше.

«Відповідність вимогам» передбачає, що вимоги повинні бути настільки чітко визначені, що вони не можуть бути зрозумілі і інтерпретовані некоректно. Пізніше, на етапі розробки, проводиться регулярне оцінки розробленого продукту, для визначення відповідності вимогам. Будь-які невідповідності повинні розглядаються як дефекти - відсутність якості. Наприклад, специфікація на певну модель радіостанції може вимагати можливості приймати певну частоту радіохвиль на відстані більш ніж 30 кілометрів від джерела мовлення. У разі якщо радіостанція не здатна виконати дану вимогу, вона не задовольняє вимоги до якості і повинна бути визнана непридатною і неякісною. Виходячи з тих же принципів, якщо Кадиллак відповідає всім вимогам до машин Кадиллак, значить це якісна машина. Якщо Шевроле відповідає всім вимогам до машин Шевроле, отже, це теж якісна машина. Ці дві машини можуть бути абсолютно різними за стилем, швидкості і економічності, але, якщо обидві вимірювати за стандартними для них наборам, тоді вони обидві будуть якісними машинами.

Визначення «придатність до використання» приймає до уваги вимоги та очікування кінцевих користувачів продукту, які очікують, що продукт або сервіс, що надається буде зручним для їх потреб. Однак різні користувачі можуть використовувати продукт по-різному, це означає, що продукт повинен володіти максимально різноманітними варіантами використання. Згідно визначення, кожен варіант використання - це характеристика якості і всі вони можуть бути класифіковані за категоріями в якості параметрів придатності до використання.

Ці два визначення якості («відповідність вимогам» і «придатність до використання») по суті рівнозначні. Різниця в тому, що варіант «придатність до використання» вказує на важливу роль вимог і очікувань замовника. Роль замовника, пов'язана з якістю, ніколи не може бути переоцінена. З точки зору замовника, якість продукту, який він придбав, складається з безлічі різних факторів, таких як: ціна, продуктивність, надійність і т.п.

Тільки ваш замовник може розповісти вам про якість, тому що це єдине що він дійсно купує. Замовник не купує продукт. Він купує ваші гарантії того, що всі його очікування до продукту будуть реалізовані. Якість - це придатність до використання. Чи робить даний продукт то, в чому маю потребу полегшує він мою роботу, чи можу я його використовувати так, як мені зручно.

Якість – з точки зору замовника або кінцевого користувача – це придатність до використання. Дане визначення бере до уваги вимоги та очікування кінцевих користувачів продуктів в тому, що продукт або сервіс, що надається буде зручний для їх потреб.

А тепер подивимося на точку зору розробника.

Якість – з точки зору розробника – це відповідність специфікованою і зібраним вимогам.

Проблема в тому, що специфіковані і зібрані вимоги - це зазвичай лише частина всіх реальних вимог і очікувань замовника. Де ж правильне визначення якості? Якість - це відповідність реальним вимогам, явним і неявним. Дуже часто неявні вимоги настільки очевидні для замовника або користувача, що він навіть не припускає, що вони невідомі розробникам. Для прикладу повернемося до наших автомобілів - замовник може детально описати вимоги до дизайну, параметрам двигуна, оформлення салону, кольором кузова, але ніде не вказати, що шини повинні бути круглими, а лобове скло - прозорим.

Замовник буде задоволений тільки тоді, коли куплений товар буде повністю задовольняти його реальним і життєвим вимогам, як специфіковані, так і немає.

За численними відгуками західних представників основним недоліком компаній, що виробляють ПО, є слабке розуміння бізнес - потреб клієнтів.

Приклад: клієнт замовив додаток для розробки і друку триколірних візитних карток. Програміст Іванов подумав, що ми живемо в століття прогресу, кольорових лазерних принтерів і триколірні візитки - відстій. Тому візитки повинні бути кольоровими. Він реалізує свої ідеї і відправляє програму клієнтові в термін. А ось тут відбувається дивна річ - клієнт у люті. Вся справа в тому, що він планував випустити на ринок програму для друку триколірних візиток, продаючи її, скажімо за, 15 доларів за диск. А потім, продавши достатню кількість дисків, запропонувати нову версію програми, яка буде робити повнокольорові візитки, і продавати її по 30 доларів, а з тих клієнтів, які купили у нього попередню версію взяти по 10 доларів за оновлення. У підсумку, клієнт вимагає переробити програму за гроші компанії і вимагає у компанії неустойку за те, що програма не була готова до терміну.

Наступний приклад: клієнт домовляється з компанією про демо-версії програми, в якій буде реалізовано ядро нової системи і частина інтерфейсу. Про терміни домовилися. Клієнт замовляє конференц-зал, запрошує журналістів, потенційних споживачів і планує презентацію демо-версії програми, все оплачує вперед. Програміст Петров працює над ядром. У якийсь момент він розуміє, що ядро можна і потрібно зробити краще. Якщо так зробити, то програма буде працювати швидше в 10 разів. На це буде потрібно всього лише 3 додаткових дні. В результаті ядро системи сяє, але до встановленого терміну не готове вікно зі звітами. У підсумку презентація провалюється, тому що вікно зі звітами є основною відмінністю програми від аналогів. Клієнт несе колосальні збитки, подає в суд на компанію і подальше виробництво продукту зупиняється. Сяюче ядро системи нікому не потрібно.

Сенс в тому, що потрібно намагатися зрозуміти, що для клієнта є найбільш важливим. Якась дрібна помилка в віконці Ебаут (About) з неправильно написаним прізвищем губернатора або мера, якого автори проекту дякують за сприяння, може привести до плачевних наслідків. І в такій ситуації якість всієї

іншої системи може виявитися неоціненим, тобто вся робота - «нанівець». Дуже важливо, щоб і програмісти, і тестери це розуміли і думали про це. Будь-які відхилення від початкових домовленостей щодо термінів, по функціональності повинні обговорюватися з клієнтом.

Якість програмного продукту характеризується набором властивостей, які визначають, наскільки продукт «добрий» з точки зору зацікавлених сторін, таких як замовник продукту, спонсор, кінцевий користувач, розробники і тестувальники продукту, інженери підтримки, співробітники відділів маркетингу, навчання і продажів. Кожний з учасників може мати різне уявлення про продукт і про те, наскільки він добрий чи поганий, тобто про те, наскільки високою е якість продукту.

Таким чином, постановка задачі забезпечення якості продукту зводиться до задачі визначення зацікавлених осіб, їх критеріїв якості і потім знаходження оптимального рішення, що задовольняє цим критеріям. Тестування е одним із способів розробки якісного програмного забезпечення і входить до набору ефективних засобів сучасної системи забезпечення якості програмного продукту.

Відома оцінка розподілу трудомісткості між фазами створення програмного продукту: 40%-20%-40%, з чого витікає, що найбільший ефект у зниженні трудомісткості може бути отриманий не тільки на стадіях тестування, а й на стадіях розробки програмного коду. Тому основний внесок до автоматизації або генерації коду потрібно здійснювати саме на цих фазах.

У цьому розділі ми наведемо деякі рекомендації, що допоможуть створювати програми, які буде легко тестувати і підтримувати в майбутньому. Дані рекомендації не претендують на абсолютну повноту і істинність — вони покликані відобразити точку зору авторів на цю тему.

Зрозуміло, що не всі і не завжди можуть або хочуть наслідувати цим і подібнім до них рекомендаціям, але до цього потрібно прагнути. Точніше, прагнути потрібно не до наслідування цих правил, а до підвищення власної майстерності, вартості своєї праці.

2. Основні поняття культури програмування

У першу чергу необхідно визначитися з термінами. Що ж таке культура і що ж таке програмування? Для обох цих понять існує велика кількість визначень, ми ж скористаємося наведеними нижче.

Під культурою розуміється набір правил (часто неформальних), стереотипів і норм поведінки.

В якості визначення поняття «програмування» ми пропонуємо лаконічне, але містке визначення: програмування це частина виробництва програмного забезпечення. Це не кодування, не трансляція ваших думок у вирази якої-небудь мови програмування, це виробництво, організація якого має бути на дуже високому рівні.

Наступне питання, на яке необхідно відповісти: для кого пишуться програми, для кінцевих користувачів чи програмістів? Комусь очевидною здається відповідь, що програми пишуться для кінцевих користувачів. Той же, хто займався підтримкою коду, написаного іншою людиною, може дуже емоційна сказати, що програми потрібно писати так, щоб їх було легко підтримувати.

Наведемо список тих «правил, стереотипів і норм поведінки», які роблять програміста культурним, а створюване ним програмне забезпечення однаково простим і легким як в тестуванні, так і у використанні.

3. Характеристики якості програмного забезпечення

Якість ПЗ має зовнішні та внутрішні характеристики. До зовнішніх характеристик належать властивості, які усвідомлює користувач програми:

1. Коректність — відсутність/наявність дефектів у специфікації, проекті та реалізації системи.
2. Практичність — легкість вивчення і використання системи.
3. Ефективність — ступінь використання системних ресурсів. Ця характеристика враховує такі фактори, як швидкодія програми і необхідний їй обсяг пам'яті.
4. Надійність здатність системи виконувати необхідні функції за певних умов; середній інтервал між відмовами.
5. Цілісність — здатність системи запобігати неавторизованому або некоректному доступу до своїх програм та даних. Ідея цілісності передбачає обмеження доступу до системи для неавторизованих користувачів, а також забезпечення правильності доступу до даних, тобто одночасну зміну взаємопов'язаних даних, зберігання лише допустимих значень тощо.
6. Адаптованість — можливість використання системи без її зміни в тих галузях або середовищах, на які вона не була орієнтована безпосередньо.
7. Правильність — ступінь безпомилковості системи, особливо щодо виведення кількісних даних. Правильність характеризує виконання системою її функцій, а не те, чи створена вона коректно. Цим правильність відрізняється від коректності.
8. Живучість — здатність системи продовжувати роботу при введенні неприпустимих даних або в напружених умовах.

Деякі з цих характеристик перекриваються, проте кожна має свої відмінні риси, які в одних випадках виражені сильніше, а в інших слабше.

Зовнішні характеристики — єдина категорія властивостей ПЗ, яка важлива для користувачів. Користувачів турбує легкість роботи з ПЗ, а не легкість його зміни. Їх хвилює коректність ПЗ, а не зручність читання або структурованість коду.

Для програмістів важливі і зовнішні характеристики ПЗ, і внутрішні. До внутрішніх характеристик ПЗ належать:

1. Зручність супроводу легкість зміни програмної системи з метою реалізації додаткових можливостей, підвищення швидкодії, виправлення дефектів тощо.

2. Гнучкість — можливий масштаб зміни системи з метою використання ii в тих галузях або середовищах, на які вона не була безпосередньо орієнтована.

3. Можливість повторного використання — масштабність і легкість використання частин системи в інших системах.

4. Зручність читання — легкість читання та розуміння вихідного коду системи, особливо на детальному рівні окремих операторів.

5. Зручність супроводу — можливий ступінь виконання блокового і системного тестування програми та перевірки її відповідності вимогам.

6. Зручність тестування — легкість розуміння системи і на рівні загальної організації, і на детальному рівні окремих операторів. Зрозумілість характеризує узгодженість системи на загальнішому рівні, ніж легкість для читання.

Як і зовнішні, деякі з цих внутрішніх характеристик якості перекриваються, але при цьому кожна з них має важливі відмітні риси.

Різниця між внутрішніми і зовнішніми характеристиками якості розмита, тому що на деякому рівні перші впливають на другі. Якщо програма не достатньо зрозуміла або незручна в супроводі, в ній важко виправляти дефекти, що в свою чергу впливає на такі зовнішні характеристики, як коректність і надійність. ПЗ, що потерпає від нестачі гнучкості, не можна поліпшити у відповідь на запити користувачів, що позначається на його практичності. Суть сказаного в тому, що одні характеристики якості полегшують життя користувачам, а інші — програмістам.

4. Стандарти та угоди

Наведемо ще одне визначення культури програмування: «Культура програмування — це те, що протистоїть хаосу». Те, що коїться в коді програми, що написана для особистого користування — це особиста справа програміста. Але виробництво ПЗ це зазвичай робота команди. І в цьому випадку те, що коїться в коді — це справа колективу.

У команді розробників має бути документ «Стандарт програмування і оформлення коду», який повинен описувати усі тонкощі програмування і оформлення коду, від «імена змінних мають бути значущими, за винятком тривіальних випадків (наприклад, лічильники циклів)», до старого принципу програміста — «кожен модуль повинен виконувати тільки одну функцію».

Цей стандарт повинен підтримуватися і постійно оновлюватися, а знати його повинен кожен, хто пише код. Це спростить читання чужого коду, чужий стиль не «різатиме око» і підвищить загальну дисципліну команди, що для програмістів особливо важливо.

Стандарт повинен включати пункт про коментарі. Коментарі мають

бути, але важливо відчувати грань і не писати коментар до кожного рядка коду. Можна використати таку стратегію коментування об'єктно-орієнтованого коду:

1. Має бути описане призначення кожного класу.
2. Кожен елемент класу так само має бути прокоментований.
3. Складні, не прозорі ділянки коду, повинні забезпечуватися коментарями.
4. Якщо через деякий час виникли проблеми з читанням тієї або іншої ділянки коду, він так само має бути прокоментований.

5. Використання засобу контролю версій

Кожен з нас може помилятися. Кожен з нас, перш ніж зрозуміє помилку, може піти настільки далеко від колись вірної ідеї, що простіше написати все заново, чим з наявного коду відновити те, що було. Для того щоб не довелося починати все заново, необхідно використати засіб контролю версій. При цьому фіксувати потрібно не лише ключові версії, але і взагалі увесь процес розробки.

Якщо розробка ведеться групою програмістів, то потрібний засіб для нейтралізації конфліктів паралельної розробки, що теж забезпечується більшістю систем контролю версій.

Багато засобів дозволяють вести паралельно дві версії коду, наприклад, першу версію, в якій виправляються помилки, виявлені при її експлуатації, і другу версію продукту.

Так само, в серйозних організаціях, використовуючи подібну систему, відповідальність за збереження коду зазвичай перекладається на адміністраторів, які успішно справляються з цією функцією.

6. Простий код і рефакторинг

Для розробника програм найціннішим ресурсом є час. У цьому підрозділі наведемо ряд порад програмістам (Стіва Макконелла, Джеффа Вогела, А. Жидкова, А. Нікітіна) для розробки хороших програм, які буде легко тестувати і підтримувати в майбутньому.

Код хорошої програми повинен бути настільки простим, наскільки це можливо. «Не потрібно ускладнювати завдання», «Keep it simple, stupid! (правило KISS)», «обирайте найпростіше з працюючих рішень», «не треба робити того, що не передбачено технічним завданням, будьте простіші».

Простий код простіший у читанні, у підтримці і тестуванні, а значить, містить менше помилок. Наприклад, навіщо застосовувати швидке або пірамідальне сортування, якщо можна обійтися сортуванням бульбашкою. Не потрібно робити щось універсальне на усі випадки життя, якщо потрібно розв'язати цілком конкретне і просте завдання.

Виникає неминуче питання — як зробити таким чином, щоб програмний код був зрозумілим, не створював складнощів при подальшому супроводі й взагалі не був нерозбірливою плутаниною?

Наведемо ряд порад програмістам для розробки хороших програм.

Коментувати програму. Необхідно завжди коментувати програмний код. Наприклад, якщо була написана процедура без супроводу ii коментарями, то коли через декілька місяців повернутися до неї для доопрацювання, відсутність коментарів призведе до додаткових втрат робочого часу.

Слід, однак, відзначити, що написання коментарів — це теж мистецтво. Для досягнення майстерності у цьому виді діяльності необхідна практика. Коментарі бувають хороші й погані. Не треба писати занадто довгих коментарів. З іншого боку, не слід писати дуже коротких коментарів.

Коментарі не тільки економлять час, вони самі потребують часу. Коментарі вимагають часу на прочитання, крім того, вони збільшують фактичні розміри програми на екрані монітора, в результаті чого перед очима може одночасно перебувати менший обсяг діючого програмного коду.

Імена для змінних, функцій/методів повинні вибиратися, виходячи з їх функціонального призначення.

Осмисллено вибирати імена змінних, методів і класів. Осмисллено вибирати імена змінних, методів і класів — це кінцева мета для написання простого програмного коду, щоб стороння людина, не маючи уявлення про те, що цей код робить, могла зрозуміти його якомога швидше.

Один з основних способів досягнення цієї мети полягає у тому, щоб давати змінним і процедурам хороші імена. При цьому необхідно дотримуватися золотої середини. Необхідно давати імена об'єктам програми досить довгі й наочні для розуміння їх сенсу, однак не настільки довгі й громіздкі, щоб це ускладнювало читання програмного коду.

Необхідно відмовитися від змінних типу kkk, або функцій на зразок gdfui(), краще ввести довгу назву getDoubleFromUnsignedInt(), проте відразу буде ясно, що відбувається.

Деякі дієслова можуть описувати практично будь-які дії всередині функцій. Наприклад, Calculation(), Handlelnput() або ProcessData(). З одного боку, назва функції може узгоджуватися з вмістом, але здогадатися про вміст за назвою практично неможливо. Тому замість Calculation() потрібно писати щось більш докладне, типу GetSolutionOfEquation().

Якщо функція щось повертає, то це повинно бути зрозуміло з назви. Приклад хороших імен: sin(), GetMaxValue(), monitor.IsReady(), fteratorNextElement().

Не обмежувати довжину імені функцій. Згідно з дослідженнями оптимальна довжина змінної — це 9-15 символів. Функція несе ще більше смислове навантаження, відповідно і довжина у неї може бути ще більшою.

Не використовувати нумерацію для функцій, які розв'язують схожі за-

вдання. Буває, що функції виконують одне і те ж завдання, але різними способами. Наприклад, ми хочемо написати функцію факторіала і реалізували два варіанти — за допомогою циклу і за допомогою рекурсії. Не потрібно називати функції Fact1() і Fact2(). Слід давати більш осмислені назви FactCicle() і FactRecursion().

Функція повинна вміщуватися на екран. Це, звичайно, в ідеалі, але дуже бажано. Якщо розмір функції складає більше двох екранів — це явна ознака того, що е сенс розділити ii на кілька частин.

Варто зазначити, що бувають винятки. Наприклад, якщо у функції е оператор switch, то вона може бути досить громіздкою.

Не використовувати для імен функцій українські слова, написані транслітом. Наприклад, назви типу risuemKolo() непривабливі, на відміну від drawCircle(). Назви з використанням англійських слів будуть нормально сприйняті будь-яким програмістом.

Обов'язково ініціалізувати змінну в місці її визначення. Це правило не дуже складне в реалізації, але вимагає уважності і дисципліни. Особливо часто спокуса не ініціалізувати змінну виникає при оголошенні цієї змінної перед розгалуженням, в якому їй має бути присвоєне значення. В хорошому випадку у всіх гілках розгалуження відбувається ініціалізація змінної. В процесі роботи над додатком, розгалуження може змінюватися, в ньому з'являються гілки, які виконуються не часто і в яких змінна не ініціалізується. Як наслідок, іноді програма поводиться невизначеним чином. Для попередження подібних ситуацій потрібно форматувати змінні в місцях їх визначення.

Уникати вкладених викликів методів. При сприйнятті нової інформації людина прагне розбити ii на пов'язані атомарні блоки, кожен з яких буде мати певний сенс. Чим менше блок інформації, тим простіше його зрозуміти. Вкладення методів збільшує розмір блоку інформації на рядок. Чим складніша функція вкладена тим більше складніші рядки коду. Досить часто зустрічаються вкладення методів, які носять більш складний зміст. Наприклад, метод, який приймає не відсортований список ключів і повертає відсортований список ключів.

У подібному коді дуже важко зрозуміти, що відбувається в поточному рядку, оскільки потрібно послідовно розгорнути великий блок інформації на сукупність дрібніших і зрозуміліших.

Налагоджувати подібний код просто неможливо, так як важко зрозуміти на якому саме етапі відбувається помилка і немає іншої можливості ознайомитись з результатами виконання вкладених функцій, крім як покроково виконати їх.

Перевіряти програму на наявність помилок. Якщо програма досить велика, в ній буде багато функцій і процедур. Як би це не здавалося клопітно, кожну функцію/процедуру необхідно перевіряти на наявність помилок. Код слід писати так, щоб він перевіряв процедуру/функцію на наявність сторонніх даних і захищався від них.

Переваги такого підходу не вичерпуються захистом програми від збоїв. Хороші механізми перевірки на наявність помилок також прискорюють налагодження. Якщо в якій-небудь процедурі наявні всі механізми перевірки, то не доведеться проходити ii крок за кроком у пошуках помилки.

Прагнути до простоти і ясності. Не треба розміщати десять рядків нормального коду в один рядок. Це призведе до неможливості швидко виправити помилку в цьому коді.

Простий код вимагає менше часу на написання, на розуміння при наступному звертанні до нього і до налагодження. Треба прагнути до простоти і ясності — це збереже вам масу часу і позбавить від непотрібних страждань.

Довжина рядка не повинна перевищувати 80 символів. Якщо необхідно переносити або викликати функцію з великою кількістю аргументів, або якінебудь складні умови в конструкціях типу if, то переносити треба обов'язково так, щоб весь код було видно без прокручування екрану по горизонталі.

Слідкувати за кількістю відступів усередині всіх конструкцій. Кількість табуляцій перед кодом повинно відповідати його вкладеності. Багато середовищ програмування зараз автоматично вирівнюють код, але не всі, так що стежити треба самому.

Рефакторинг — це процес зміни коду без зміни функціональності. Тобто з точки зору користувача програма не міняється. Але з точки зору програміста вона міняється, при чому в кращу сторону, якщо рефакторинг проводиться грамотно.

Рефакторинг полягає в чищенні програмного коду від «заплаток», спрощенні програмного коду і зміни архітектури в цілях зробити його зрозумілішим, гнучкішим тощо. Подібне чищення необхідно проводити регулярно.

Іншим хорошим кандидатом на рефакторинг є код, який був написаний без попереднього проектування або при неповному розумінні завдання. Отримавши досвід розв’язання задачі, можна зробити код програми якіснішим.

Код, який кілька разів переписувався або змінювався нашвидкуруч, також може вимагати рефакторинга. Відразу написати ідеальний код — це дуже складне завдання. Код повинен еволюціонувати, і в цьому допоможе рефакторинг. Більшість програмістів використовують чужий код. Це дозволяє заощадити час на розробці, але при цьому разом з кодом приходять не тільки його проблеми, а й стиль, яким він написаний.

При включенні подібних рішень е сенс подумати про їх попередній рефакторинг, так як наступні виправлення коду, реалізованого на основі цього «компонента», зажадають набагато більше трудовитрат.

Рефакторинг необхідний, якщо:

1. Мас місце складність модифікацій.
2. Підтримка вимагає все більше і більше ресурсів.
3. Має місце складність читання.
4. С коментарі типу: «Як це працює — не зрозуміло».
5. Використовується чужий код.

7. Управління ресурсами

Управління ресурсами — це тема багатьох вже написаних книг. Тому ми тільки торкнемося декількох ключових моментів, не поглиблюючись в подробиці.

Культурний програміст не залишить за собою десяток відкритих файлів, кілька підключень до БД і не звільнену пам'ять, оскільки ПЗ, яке «вішає» усю систему і блокує половину файлової системи, не можна назвати хорошим.

Програма повинна захоплювати рівно ті ресурси, які їй потрібні в даний момент, і звільняти їх відразу, після того, як потреба в них задоволена. Сюди ж входять і виняткові ситуації. Якщо у вас обірвалося підключення до БД, це зовсім не привід залишати заблокованим локальний файл.

Так само варто відмітити той факт, що зазвичай, з метою оптимізації, ресурс захоплюють один раз і тримають його до тих пір, поки він може вимагатися. Але якщо ж ресурс є критичним, то його необхідно постійно захоплювати і звільняти.

8. Оптимізація

Перед початком розробки необхідно відповісти на питання: «З якого моменту необхідно починати працювати над оптимізацією?». Існують підтверджені часом крилаті фрази: «90% часу витрачається на 10% коду» (які ще потрібно виявити) і «Спочатку змусьте програму працювати, потім змусьте її працювати правильно і тільки потім змусьте її працювати швидко».

Необхідно вибирати архітектурні рішення серед шаблонів проектування, добре проаналізувавши наслідки того або іншого вибору і займатися оптимізацією тільки після того, як з'явиться така необхідність.

Оптимізація — це ворог ясності. Слід, однак, відзначити, що у деяких випадках оптимізація необхідна. Це особливо справедливо для ігор. Проте, і це дуже важливо, майже ніколи не відомо заздалегідь, що саме необхідно оптимізувати, до тих пір, поки не буде протестований реально функціонуючий код за допомогою інструменту під назвою профайлер. Профайлер — це програма, яка спостерігає за програмою і оцінює час, що витрачається на виконання окремих ділянок коду.

Необхідно спочатку написати ясну і працюючу програму, а тільки потім займатися оптимізацією.

Нарешті, наведемо таке правило: «Для того щоб заощадити час потрібно витратити пам'ять.

Для того щоб заощадити пам'ять потрібно згаяти час». Це означає, що проміжні дані можна або зберігати в пам'яті (заощадження часу), або всякий раз обчислювати (економія пам'яті).

9. Діалог з користувачем

Хороше програмне забезпечення мас бути швидким. Користувач не повинен гадати, чи то він кнопку не натиснув, чи то мишка не спрацювала, чи то програма все-таки щось робить.

Якщо потрібно виконати тривалу операцію, то необхідно запускати ii в окремому потоці, тоді як в основному повідомляйте користувача про початок виконання операції, про хід виконання і реагуйте на його дії.

З одного боку, програмне забезпечення, принаймні, його інтерфейсна частина, повинна постійно вести активний діалог з користувачем. Хороше ПЗ не «зависає» без відповідного повідомлення і не робить того, чого користувач не замовляв.

З іншого боку, коли якась часта операція супроводжується питанням, на яке постійно потрібно відповідати, — це теж дратує користувача.

Так само варто відмітити, що діалог повинен здійснюватися грамотно і зрозумілою мовою. Повідомлення, що виводяться програмою, мають бути максимально повними і зрозумілими, але і не містити зайвої інформації. Не потрібно виводити зайвих повідомлень.

Програма веде діалог з користувачем засобами інтерфейсу. А створення якісного інтерфейсу і побудова хорошого діалогу — це окрема наука (юзабіліті, зручність використання), яка мало перетинається з програмуванням. Тому призначений для користувача інтерфейс хорошого ПЗ розробляється не програмістами, а дизайнерами призначеного для користувача інтерфейсу. Звичайно, можливо, що хороший програміст також е і хорошим дизайнером, але у більшості випадків це не так.

10. Система довідки

Про всяк випадок відмітимо, що хороше ПЗ супроводжується хорошою документацією і посібником користувача. Але тут мова піде не про це. У цьому підрозділі говоритися про те, що кожен елемент (блок, вікно, відеокадр і тому подібне) повинен мати кнопку «Довідка».

При натисненні цієї кнопки користувач повинен отримати вичерпну інформацію про призначення елементу в цілому і в кожній його частині окремо, про те як користуватися тим або іншим елементом.

Довідка форми введення повинна містити приклади даних, що вводяться. При тому не лише коректних, але і не коректних. Довідка форми виведення повинна описувати усі дані, що виводяться. Природно, що довідка має бути написана не менш хорошою мовою, ніж діалоги.

11. Обробка помилок

Помилки можна розділити на три види: помилки користувача, помилки

програміста і помилки устаткування. Усі три види помилок хороше ПЗ повинно грамотно обробляти.

Помилки користувача найчастіше зводяться до різних друкарських помилок і нерозуміння призначення елементів управління. Такі помилки не повинні ставати причинами програмних помилок: дані, що вводяться, повинні перевірятися, і якщо виявлена помилка, то має бути надана можливість для її виправлення. При тому усе це повинно бути добре прокоментовано. Не можна просто припиняти виконання операції, користувач повинен зрозуміти, чому операція не виконана. Намагайтеся давати користувачеві рекомендації щодо виправлення помилок.

Обробка помилок програміста включаючи їх виявлення (перехоплення), спробу виправлення і відновлення роботи програми. Якщо помилка виправлена, то не потрібно повідомляти про неї користувачеві. Якщо виправити не вдалося, то необхідно повернути програму в коректний стан і повідомити користувача, в який стан програма повернена. Якщо помилка критична, то необхідно акуратно завершити програму, повідомивши користувача про причини завершення і дати йому можливість зберегти якомога більше даних.

Помилки програміста включають і непередбачену поведінку програми. Тому в точках галуження необхідно враховувати непередбачені варіанти. Наприклад, у блоці switch завжди має бути мітка default, яка зупиняє виконання операції з відповідними повідомленнями користувача.

Помилки устаткування зазвичай виправити програмними засобами неможливо. Тому кращою стратегією буде повідомлення про неї користувачеві, відмінити операцію, якщо програма може продовжити роботу або коректне завершення в іншому випадку.

12. Юніт-тести

Хороше ПЗ на передбачених вхідних даних, завжди працює правильно. Але це не означає, що потрібно писати програми без помилок — так не буває. Але потрібно бути упевненим, що передбачена поведінка працює без помилок.

Якщо функція учора працювала коректно, то це зовсім не означає, що вона і сьогодні працює коректно. У складних застосуваннях, із складними зв'язками буває дуже важко передбачити наслідки тієї або іншої зміни.

Для того щоб не гадати «А чи все правильно працює?», існують юніт-тести. Вони дозволяють відносно швидко (на багато швидше, ніж вручну) перевірити працездатність усієї передбаченої функціональності і переконатися, що, виправивши одну помилку, було не зроблено іншу.

Краще всього, якщо час виконання дозволяє, при тестуванні поточної роботи виконувати тести усієї системи або модуля, щоб відразу виявити помилки, що з'явилися в інших ділянках коду. Якщо ж час не дозволяє, то юніт-тести необхідно запускати як мінімум один раз в день і обов'язково перед інтеграцією (з фіксацією в систему контролю версій).

13. Субкультура

Практично кожна область програмування має свої культурні особливості. Наприклад, існують такі поняття як «Культура паралельного програмування», «Культура мережевого програмування» тощо. Необхідно вивчати культуру тієї області, в якій працюєте. Це заощадить вам багато часу і сил.

Тут же можна сказати і про різні мови. Наприклад, в C і C++ широко розвинена культура роботи з пам'яттю і покажчиками, тоді як в Java і C#, за рахунок «збирача сміття», значення навичок роботи з пам'яттю сильно ослаблене.

Існують також мови, які підштовхують до безкультурності або, навпаки, до підвищення культури програмування. Це залежить від завдань, які зазвичай розв’язуються з використанням тієї або іншої мови програмування.

До розв’язання складних завдань залучаються висококваліфіковані фахівці, які задають високу культуру програмування. Відповідно, якщо мова використовується для розв'язання складних завдань, то і колектив розробників, зазвичай, складається з висококваліфікованих фахівців, які підвищують загальну культуру всього колективу розробників.

Контрольні запитання

1. Які властивості якості належать до зовнішніх характеристик?
2. Які властивості якості належать до внутрішніх характеристик?
3. Для чого використовується засіб контролю версій?
4. Наведіть характеристики простого коду.
5. Що таке рефакторинг програмного коду?
6. Для чого існують юніт-тести?