Забезпечення якості, метрики якості

Містить наступний матеріал:

Вступ

короткий аналіз попереднього матеріалу

Якість програмного коду

Форматування та оформлення коду

Правила іменування

Розмір структурних блоків

Документованість коду

Загальноприйняті стилі оформлення коду

Характеристики якості ПЗ

Наведено перелік та зміст характеристик якості

Моделі якості програмного забезпечення

Модель МакКола

Модель Боема

Модель FURPS/FURPS+

Модель Гецці

Модель якості Дромі

Модель якості SATC

Модель якості ISO 9126

Модель якості QMOOD

Багаторівневий підхід до моделей якості.

Моделі оцінки надійності

Класифікація моделей

Модель Шумана

Модель Міллса

Модель Джелінскі-Моранди

Модель Липова

Метрики якості програм

Означення

Метричні шкали

Метрики складності

Метрика Холстеда

Метрика Маккейба

Метрика Майєрса

Метрика Джилба

З чого розпочати забезпечення якості

Питання для самоконтролю

Вступ

На минулому занятті ми з вами встановили, що в процесі розробки ПЗ на тестування покладається завдання *забезпечення якості ПЗ* як головного завдання тестування*.*

Разом з тим ми встановили, що процедура тестування в водоспадній моделі розробки (та їй подібних), безпосереднього впливу на якість не має. Причиною є відокремлення процесу тестування від процесу проектування та кодування. Тож в межах згаданих моделей розробки тестування може виконувати тільки *функцію контролю якості.*

Ситуація міняється в моделях гнучких технологій (ітераційного характеру), де тестування виконує *функцію зворотного зв’язку* з програмістом відносно стану якості програми і таким чином сприяє усуненню дефектів в процесі розробки.

Завдячуючи зміні технологічного процесу змінилась і роль тестувальника в ньому. Отже, ключ до забезпечення якості продукту слід шукати в *організації якісного технологічного процесу розробки*, за що має відповідати Quality Assurance Engineer.

Є ще один висновок, який слід запам’ятати – *ціна помилки по мірі просуванню розробки в часі зростає в експоненціальній залежності*. Тож ніякі зусилля кваліфікованих спеціалістів не усунуть дефектів, спричинених помилками на стадії формування вимог.

Це і дає відповідь на питання *з чого слід починати тестування* для забезпечення якості ПЗ - з тестування вимог до ПЗ.

**Примітка**. В подальшому ми будемо розрізняти ці два поняття – помилка і дефект. *Дефект* – це невідповідність між очікуваною і реальною поведінкою програми, а *помилка* – це те, що спричинило появу дефекту. Дефект – це об’єкт ‘полювання’ тестувальника, а помилка – відповідно, об’єкт виправлення розробника (системного аналітика, програміста, тощо).

Тепер ми підійшли до головної цілі нашого заняття – як ми можемо забезпечувати якість, якщо ми не знаємо, що це таке.

Розглянемо визначення «якість програмного забезпечення» у контексті міжнародних стандартів. В стандарті

[[1061-1998 IEEE Standard for Software Quality Metrics Methodology](https://ieeexplore.ieee.org/document/749159)]

зазначено, що «якість програмного забезпечення — це стан, у якому програмне забезпечення має необхідну комбінацію властивостей».

Це визначення вимагає пояснення що значить «необхідну комбінацію властивостей»

Відповідно ло стандарту це визначення дещо відрізняється

[[ISO 8402:1994 Керування якістю та забезпечення якості](https://files.stroyinf.ru/Data1/5/5812/index.htm)].

Якість програмного забезпечення - це сукупність *характеристик ПЗ*, що належать до його здатності задовольняти встановлені та очікувані потреби.

Якщо оцінки зовнішньої якості підтримуються стандартами, то для оцінки та контролю якості програмного коду (внутрішньої якості) державних стандартів немає. Кожен розробник розуміє якість по-своєму, з досвіду. Кожна команда для окремих проектів оцінює код по-своєму. Команда оновлюється, розробники йдуть, тимліди змінюються - визначення якості змінюється.

Проблема… І її треба розв’язувати, оскільки витрати на супровід cуттєво залежать від якості коду. Це поле внутрішньої стандартизації команди розробників.

Який код легко супроводжувати ? В першу чергу – це такий, який легко і приємно читати. Який не залишає проблем ні розробнику, ні тімліду, ні тим, хто програму супроводжує. Це, так званий, чистий код.

Чистий код, поняття загалом субєктивне. Якщо щось комусь добре, то іншому воно може виявитись поганим. Проте на сьогодні є таке розуміння чистого коду, з яким погоджується більшість програмістів. Це розуміння є змістом наступного розділу

Якість програмного коду  
*Поняття «якість програмного продукту» включає не тільки повноту і коректність реалізації необхідного функціоналу, а й простоту підтримки та модифікації програми. Як же убезпечити себе та колег від кошмару підтримки нечитабельного ?*

При створенні будь-якої програми розробники турбуються насамперед про її працездатність, адже якщо ПЗ не задовольняє вимогам замовника, то все останнє не має ніякого значення.

Однак випуском працюючої версії життєвий цикл не обмежується. У період експлуатації програми в ньому виявляються помилки, які необхідно виправляти, а у користувачів можуть з'являтися нові вимоги, так що програму доводиться доопрацьовувати та випускати нові версії.

Тут і постає питання про те, наскільки трудомістким є внесення змін до вже написаного коду, який ніхто не чіпав місяці, а то й роки. Одна річ – виправляти програму «за гарячими слідами», доки всі учасники розробки добре представляють її структуру та схему роботи, і зовсім інша – модифікувати незнайомий код, у якому ще треба розібратися. Практика показує, що нерідко програмісти важко продираються крізь нетрі навіть власних творів, створених кілька років тому. Що вже казати про ситуацію, коли доводиться правити чужі програми.

Сукупність характеристик коду, що впливають на трудомісткість його сприйняття та внесення змін до нього, прийнято називати якістю коду. Чим вище якість коду програми, тим простіше його підтримувати та оновлювати. Як наслідок, розробники можуть зосередитися на реалізації справді нового функціоналу, а не займатися створенням нового велосипеда лише тому, що не змогли полагодити педаль у старого.

Як же убезпечити себе та колег від кошмару підтримки нечитабелього коду?  Нижче ми постараємося розглянути основні фактори, що впливають на якість коду, а також узагальнити рекомендації щодо його покращення, що застосовуються у різних проектах та компаніях.

Форматування та оформлення коду

Одним із головних аспектів якості коду є його візуальне оформлення, за рахунок якого можна суттєво підвищити читабельність програми і вже тим самим серйозно полегшити її підтримку та доопрацювання.

Першим моментом правил оформлення коду є стиль відступів для відокремлення структурних блоків програми – тіл функцій, циклів тощо. Для відступів використовуються символи табуляції або пробілу, і якщо використовуються пробіли, скільки їх (як правило, вибір між значеннями 2, 4 і 8).

Здавалося б, необхідність використання одноманітного стилю відступів принаймні в рамках одного модуля очевидна, але в реальному житті часто зустрічаються змішування стилів, що дуже ускладнюють читання програми.

Типовий випадок: якщо в одних місцях для відступів використовуються чотири пробіли, а в інших – табуляція, то в текстовому редакторі, у якого ширина табуляції дорівнює чотирьом пробілам, нічого кримінального не буде помітно. Але якщо відкрити такий код у редакторі, де табуляція дорівнює восьми пробілам, то вийде неабияка каша. Тому не варто задовольнятися тим, що у вашому IDE код виглядає красиво, необхідно забезпечити його читання незалежно від налаштувань редактора.

Правила форматування коду включають і низку інших аспектів, однаковість яких впливає на сприйняття програми:

* використання прогалин поруч із дужками та операторами в логічних та арифметичних виразах (наприклад, чи відокремлювати в конструкції «i=1» знак рівності пробілами чи ні);
* правила розміщення фігурних дужок, що обмежують структурні блоки програми в багатьох мовах (наприклад, чи слід у програмі мовою С ставити відкриваючу фігурну дужку на тому ж рядку, де розташовується логічна умова «if», або потрібно переносити її на новий рядок);
* правила розбиття логічного виразу з багатьма умовами на кілька рядків;
* вимоги до вертикального вирівнювання, щоб кілька поспіль подібних інструкцій, що йдуть, виглядали красиво:

**int son = 0;**

**int daughter = 1;**

**int father = 2;**

* та інші моменти, які часто специфічні для конкретних мов програмування.

Правила іменування

Крім власне форматування, простота сприйняття коду багато в чому залежить від способу іменування змінних, функцій, членів класів та інших сутностей. Абеткова істина, що імена мають бути «розмовляючими», знайома всім, але при її реалізації на практиці між розробниками виникають різночитання. Почнемо із запитання: наскільки «розмовляючими» мають бути імена?

На зорі програмування, та й спочатку розвитку мов високого рівня, назви давали по можливості короткі.  Однак прогрес не стоїть на місці, екрани моніторів у розробників тепер зазвичай вміщують більше 80 символів (навіть якщо ви набираєте код на смартфоні), а IDE надають такі зручні можливості, як індексування та автодоповнення. Не дивно, що довжина імен збільшилася і ідентифікатори стали більш інформативними

Втім, занадто старатися не варто, назва не повинна перетворюватися на цілу повість. Як правило, імена функцій та глобальних змінних виходять довшими, імена локальних змінних коротші.

Слід уникати наявності декількох довгих імен, що відрізняються лише парою літер у середині, адже при побіжному погляді на код такі змінні можуть виявитися однаковими.

Не варто забувати і про деякі класичні назви - наприклад, зазвичай можна без коментарів використовувати "i" як лічильник циклу, називати методи класу, що повертають або встановлюють значення деякого поля Field - getField () і setField () відповідно і так далі.

Бажання використовувати багатослівні назви цілком логічно (особливо для функцій і методів), але перед програмістами-початківцями традиційно постає питання: як розділяти слова в назві?

Приклади іменування в [POSIX](http://www.opennet.ru/man.shtml?list=3) - «UpperCamelCase» та «lowerCamelCase»). Іноді можна зустріти як роздільник дефіс, але у багатьох мовах цей символ не можна використовувати в імені ідентифікатора.

Необхідно відзначити використання у багатьох проектах так званої [угорської нотації](https://uk.wikipedia.org/wiki/%D0%A3%D0%B3%D0%BE%D1%80%D1%81%D1%8C%D0%BA%D0%B0_%D0%BD%D0%BE%D1%82%D0%B0%D1%86%D1%96%D1%8F), яка передбачає додавання до імені змінної префікса для ідентифікації її типу або семантичного значення.

Наприклад, рядки можуть передуватися префіксом "s" (sMyName), покажчики - префіксом "p" (pMyVariable) і так далі.

Використання констант

Крім змінних, у більшості програм використовують і константні значення, незмінні під час роботи. Однак використовувати їх у коді безпосередньо слід з обережністю. Натомість слід оголосити константу з осмисленим ім'ям (якщо мова програмування не дозволяє визначати константи явно, то можна обійтися і простою змінною) і використовувати це ім'я.

Адже у людини, яка переглядає код, може виникнути питання: чому використовуються саме це число, рядок чи інший вираз і звідки воно? Звичайно, можна це вказати в коментарі, але якщо фіксовані значення мелькають у багатьох місцях, то коментувати їх навряд чи розумно.

Одним із вагомих аргументів, які часто наводяться на користь іменованих констант, є простота зміни їх значень – у разі такої необхідності вам достатньо змінити код тільки в одному місці, а не лазити по всьому проекту та визначати місця, які треба оновити. Але навіть якщо ви впевнені, що константа ніколи не зміниться, все одно має сенс використовувати для неї певний ідентифікатор.

Класичним прикладом тут є число Пі - єдиним випадком, коли вам може знадобитися "перевизначити" його значення, є зміна точності (тобто якщо ви раптом вирішите, що "3.14" недостатньо для розрахунків і треба "3.14159"). Якщо ж відкинути можливість виникнення такої ситуації, то число Пі навряд чи зміниться.

Той, хто переглядає програму і розуміє, що вона робить, скажімо, тригонометричні розрахунки, напевно здогадається, що ховається за числом, що повсюдно використовується, «3.14». Однак чи зможе він розпізнати, що за «1.047» ховається Пі/3? Можна, звісно, ​​писати «3.14/3». Але погодьтеся, «Pi» та «Pi/3» коротші, зручніші та наочніші.

Розмір структурних блоків

Розробники давно дійшли висновку, що для кращої керованості та зрозумілості код програми краще розбивати на відносно незалежні блоки – наприклад, на процедури та функції у процедурній парадигмі програмування або на класи та їх методи у ОВП. Але наскільки дрібними чи великими мають бути тіла функцій та методів?

Частково відповідь це питання дають самі парадигми, багато з яких рекомендують створювати самодостатні, але водночас прості модулі. Наприклад, філософія UNIX має на увазі створення програм, які роблять щось одне, але роблять це добре.

Той самий підхід цілком застосовний до функцій і методів класів – кожна функція має виконувати одне завдання, і навіть її поведінка має визначатися виключно її аргументами, а чи не купою зовнішніх умов на кшталт глобальних змінних.

Традиційна рекомендація з візуальної точки зору полягає в тому, що тіло підпрограми має поміщатися в екран монітора, щоб її можна було охопити поглядом. Зрозуміло, що ця вимога досить умовна – висота екрана у різних програмістів різна, та й розмір коду з тією самою функціональністю різними мовами може мати різний обсяг.

Слід враховувати інші оцінки складності функцій і методів, наприклад, кількість локальних змінних. Відомо, що середньостатистична людина здатна маніпулювати одночасно приблизно 7-ма сутностями. Тож якщо функція оперує парою десятків змінних, більшість людей навряд чи швидко зрозуміють суть її роботи.

Те саме відноситься і до рівня вкладеності структурних блоків - при великій кількості вкладених розгалужень функція сприймається досить складно.

Документованість коду

Широко відомо таке напівжартівливе твердження, що добре написаний код коментарів не потребує – він очевидний сам по собі. Звичайно, до такого стану речей варто прагнути, але далеко не завжди це виходить, і без текстових підказок «людською» мовою не обійтися.

Втім, надмірне коментування також не є гарною практикою – погодьтеся, не дуже зручно вивчати код, кожні два рядки якого передують десятку рядків коментарів.

Поширений принцип при написанні коментарів полягає в тому, що більшість з них повинні описувати, що робить та чи інша ділянка програми, а не як вона це робить. Як наслідок - збиткова кількість коментарів забезпечена.

Окремо хотілося б наголосити на необхідності опису вхідних параметрів та середовище, в якому має працювати функція. Також обов'язково супроводжуйте коментарями код модулів, в умовах роботи яких ви сумніваєтеся. Це може заощадити багато зусиль вам і вашим колегам у майбутньому, коли виявиться, що код, що працював роками, раптом став видавати помилки.

Коментарі в коді часто є основою документації до програми. Широкого поширення набули системи на зразок Javadoc і Doxygen, що автоматично генерують документацію на основі програмного коду та коментарів.

Наприклад, подібним чином можна автоматично генерувати документацію на API-бібліотеки – з вихідного коду виокремлюється синтаксис функцій, класів і змінних, що експортуються, а відповідні коментарі стають їх описом.

Використання подібних інструментів вимагає оформляти код строго певним чином (щоб інструмент міг розпізнати описи кожного параметра функції, значення, що повертається і так далі).

Тому побічним ефектом їх застосування є одноманітний стиль коментарів, що також йде на користь читання.

Зрештою, одна з головних вимог до коментарів полягає в тому, що вони мають відповідати коду. На жаль, не завжди після серйозної зміни ділянки програми розробники спроможні відповідним чином оновити коментар, що його описує. І згодом у людини, яка вивчає програму, постає логічне питання: це коментар застарілий або в реалізації є помилка?

Дозволити цю дилему не завжди просто. Тож відсутність коментаря – погано, але невірний коментар – ще гірший.

Загальноприйняті стилі оформлення коду

Вище ми розглянули основні аспекти, які зазвичай фіксуються у правилах оформлення коду. Для багатьох аспектів однозначних рекомендацій немає, але є кілька альтернативних підходів. Кожна альтернатива має свої переваги і недоліки, і багато в чому перевага того чи іншого підходу – питання естетичне і залежить від конкретної людини.

Наприклад, угорська нотація спочатку з'явилася і активно використовувалася в Microsoft (зокрема, в MFC), а ось творець ядра Linux Лінус Торвальдс вважає її неповноцінною та надмірною.

Не останню роль може відігравати якою мовою програмування ви користуєтесь. Творці деяких мов постаралися зменшити кількість аспектів, через які можливі розбіжності при форматуванні – наприклад, у Python не використовуються фігурні дужки чи інші явні розмежувачі структурних блоків, а межі блоків визначаються виключно за вирівнюванням тексту. Це знімає частину протиріч і змушує вирівнювати код, але простір для суперечок все одно залишається – наприклад, чи використовувати для відступів пробіли чи табуляцію?

Автори багатьох сучасних мов самі розробляють для своїх продуктів рекомендації щодо оформлення написаних на них програм. Наприклад, існують офіційні рекомендації для [Java](http://www.oracle.com/technetwork/java/codeconv-138413.html), [C#](http://msdn.microsoft.com/en-us/library/ff926074.aspx) та інших мов. У цих стандартів можливі деякі варіації від команди до команди, але такі відмінності зазвичай мінімальні і до них можна швидко адаптуватися.

Деякі відмінності цих стилів можна почитати у Вікіпедії [[тут](https://en.wikipedia.org/wiki/Indentation_style)]. До речі, нерідко прихильники кожного конкретного стилю дуже негативно ставляться до альтернативних підходів. Майте це на увазі, коли вливатиметеся в нову команду!

Нарешті, у багатьох компаніях та проектах стиль оформлення коду є обов'язковою частиною загальніших угод про кодування (Coding conventions). Крім власне оформлення, вони можуть включати правила організації файлів у проекті (наприклад, вимагати дотримання принципу «один клас – один файл» в ОВП), рекомендації щодо використання тієї чи іншої парадигми програмування, алгоритмів, шаблонів проектування тощо.

У сукупності ці угоди покликані підвищити якість отримуваних програмних продуктів, і саме з ними вам з великою ймовірністю доведеться ознайомитись перед тим, як приступати до роботи на новому місці.

Отже, добре оформлений і читаний код - необхідний фактор простоти підтримки програмного продукту, який істотно полегшує життя розробників. Адже програміст – досить мобільна професія, представникам якої часто доводиться переходити з однієї команди в іншу і перемикатися з проекту на проект. Відповідно одна і та сама людина за свою професійну кар'єру може взяти участь у безлічі проектів, а над одним і тим же кодом за час його життєвого циклу може попрацювати велика кількість програмістів.

Дотримання стандартів істотно полегшує ці процеси, дозволяючи людям відразу концентруватися безпосередньо на роботі, а не на вивченні коду, що дістався у спадок, і притирання до стилю попередників і товаришів по команді.

При роботі в команді необхідно дотримуватись єдиного стилю, навіть на шкоду власним уподобанням. Якщо ви приходите в проект або компанію з багатою історією, то вибирати, швидше за все, не доведеться - вас просто поставлять перед фактом, який саме набір правил слід використовувати.

А ось у навчальних командних проектах (якщо такі передбачені у вашому навчальному закладі) такі правила необхідно виробити самостійно перед початком написання коду. Однак брати ці вимоги зі стелі не варто – для всіх популярних мов програмування вже є більш-менш стандартні правила, на них треба ґрунтуватися. Відмінно сформулювали розробники Google у своєму C++ Style Guide – «дотримуйтесь здорового глузду і будьте послідовні».

Характеристики якості ПЗ

**Функціональність** — визначається як здатність програмного забезпечення вирішувати завдання, які відповідають встановленим та очікуваним потребам користувача за встановлених умов використання програмного забезпечення.

Тобто ця характеристика гарантує, що програмне забезпечення працює правильно і точно, є сумісним з її оточенням, відповідає галузевим стандартам та захищено від злому.

**Надійність** — здатність ПЗ виконувати передбачені специфікаціями завдання у заданих умовах з заданою ймовірністю відмов протягом зазначеного часу.

Атрибути цієї характеристики є збереження повноти і цілісності усієї системи, здатність самостійно та коректно відновлюватися після збоїв (відмовостійкість).

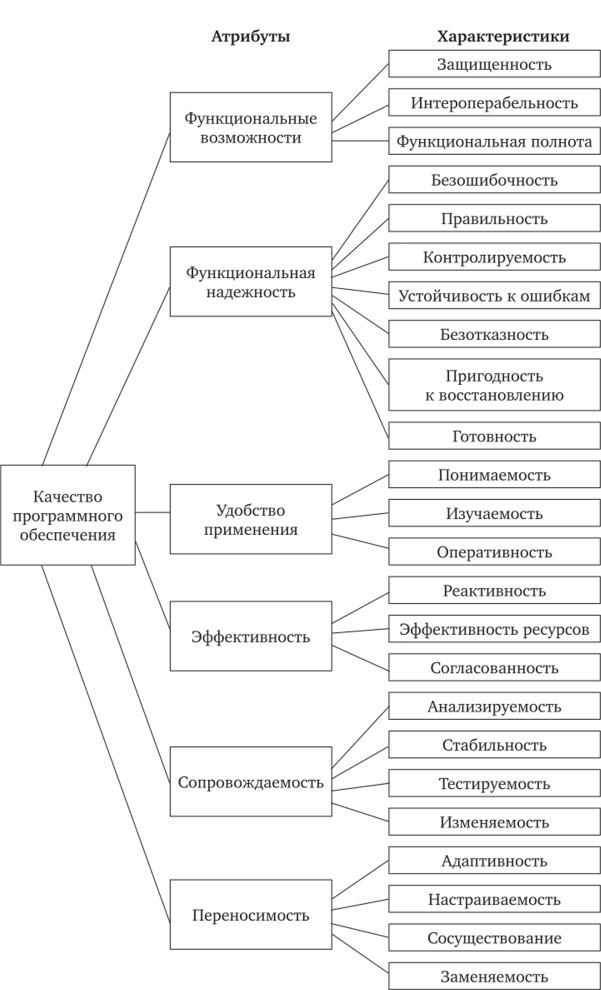
**Зручність використання** – дозволяє легко зрозуміти, вивчити та використовувати ПЗ і вилядає привабливо для користувача.

**Ефективність** — здатність програмного забезпечення надавати необхідний рівень продуктивності відповідно до виділених ресурсів, часу та інших заданих умов.

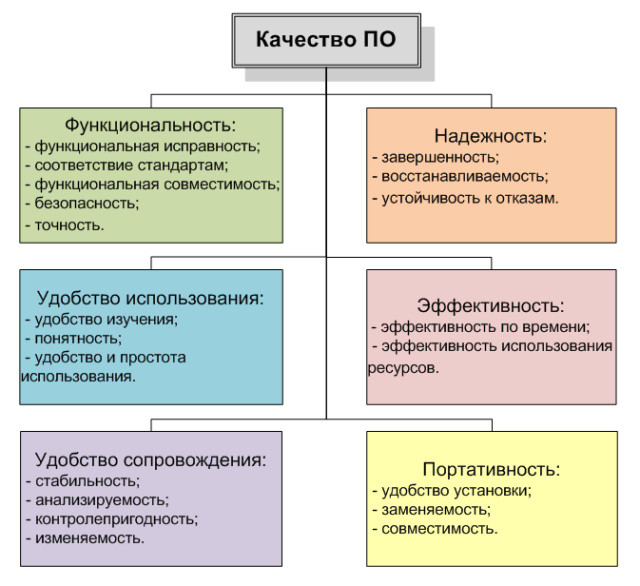
**Ремонтопридатність** – легкість, з якою програмне забезпечення може бути проаналізоване, протестоване та змінене при виявленні дефектів, реалізації нових вимог, полегшення майбутнього обслуговування та адаптації до заданого середовища (внутрішня якість ПЗ).

**Переносність** (кросплатформність)– здатність до простого перенесення ПЗ з одної програмно-апаратної платформи на іншу.

З властивостями познайомились. Тепер розберемось як оцінювати якість. Для цього існує декілька моделей оцінки.



Моделі якості програмного забезпечення



**Рис.1 –  Модель якості програмного забезпечення  (ІSO 9126-1)**

Найбільш поширеною є багаторівнева модель якості програмного забезпечення, представлена ​​в наборі [стандартів ISO 9126](https://ru.wikipedia.org/wiki/ISO_9126). На верхньому рівні виділяють 6 основних характеристик якості ПЗ, кожна з яких визначається набором атрибутів з відповідними метриками для подальшої оцінки (рис. 1).

Загалом ця модель виділяє набір якісних оцінок якості, проте користувачі ПЗ часто потребують оцінки якості як якісно, ​​так і кількісно.

Моделі якості, які є в даний час, в більшості випадків є ієрархічними моделями в залежності від прийнятих критеріїв якості і пов'язаних з ними метрик.

Усі моделі якості можуть бути поділені на три категорії відповідно до методів, на основі яких вони були створені.

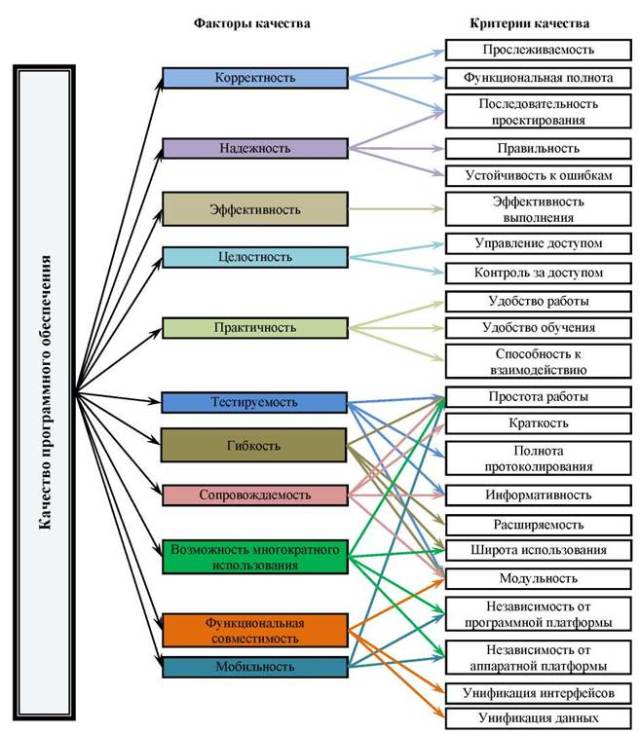
До першого виду можна віднести теоретичні моделі, що ґрунтуються на попередньо прийнятій гіпотезі відношення між змінними якості.

До другого виду належать моделі «управління даними», що ґрунтуються на статистичному аналізі.

І нарешті, комбінована модель, у якій інтуїція дослідника використовується для визначення потрібного виду моделі.

Моделі якості спрощують оцінку якості.

**Модель МакКола**

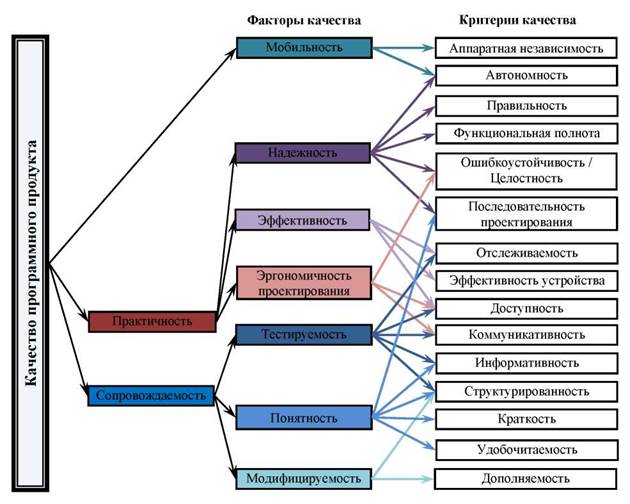


**Рис 2. Модель якості МакКола**

Модель якості МакКола (див. рис. 2) має три головні напрямки для визначення та ідентифікації якості програмного забезпечення:

* використання (коректність, надійність, ефективність, цілісність, практичність);
* модифікація (тестованість, гнучкість, супроводжуваність – фактори якості важливі для розробки нової версії програмного забезпечення);
* переносимість (мобільність, можливість багаторазового використання, функціональна сумісність - фактори якості важливі для переносимості програмного продукту на інші апаратні та програмні платформи).

**Модель Боема**



**Рис 3. Модель якості програмного продукту Боема**

Модель якості Боема представляє характеристики програмного забезпечення у більшому масштабі, ніж модель МакКола. Модель Боема схожа на модель якості МакКола тим, що вона також є ієрархічною моделлю, структурованою в ієрархії високорівневих, проміжних і примітивних характеристик, кожна з яких робить свій внесок у рівень якості програмного забезпечення.

Модель є досить практично. Вона описує, як легко, надійно та ефективно програмний продукт може бути використаний та супроводжений, характеризує наскільки легко змінити та повторно протестувати програмний продукт. Мобільність представлена тим, як програмний продукт може використовуватися при зміні програмних та апаратних засобів.

**Модель FURPS/FURPS+**

Акронім FURPS, що використовується в позначенні моделі, означає наступні категорії вимог до якості ПЗ:

* Functionality (Функціональність) / особливості, можливості, безпека /;
* Usability (Практичність) /людський фактор, ергономічність, документація користувача/;
* Reliability (Надійність) /частота відмов, відновлення інформації, прогнозованість/;
* Performance (Продуктивність) /час відгуку, пропускна спроможність, точність, доступність, використання ресурсів/;
* Supportability (Експлуатаційна придатність) /тестованість, розширюваність, адаптованість, супроводжуваність, сумісність, конфігурованість, обслуговування, вимоги до встановлення, локалізованість/.
* Символ «+» розширює модель FURPS, додаючи до неї:
* обмеження проекту (обмеження за ресурсами, вимоги до мов та засобів розробки, вимоги до апаратного забезпечення);
* інтерфейс (обмеження, що накладаються на взаємодію із зовнішніми системами);
* вимоги до виконання,
* фізичні вимоги
* вимоги до ліцензування.

FURPS модель якості дещо схожа з моделями МакКола та Боема, але на відміну від них складається з двох шарів, перший визначає характеристики, а другий - пов'язані з ними атрибути.

Основною концепцією, що лежить в основі моделі якості FURPS, є декомпозиція характеристик програмного забезпечення на дві категорії вимог, а саме,

функціональні (F) і нефункціональні (URPS) вимоги.

Зміст цих двох категорій може бути використанй як вимоги до програмного продукту, так для оцінки якості ПЗ.

На даний час модель FURPS+ широко використовується в розробці програмного забезпечення і при ідентифікації вимог до системи доцільно використання FURPS+ моделі як універсального контрольного переліку характеристик продукту.

**Модель Гецці**

В цій моделі чітко розрізняється якість продукту та процесу. Відповідно до моделі Гецци до якості програмного забезпечення відносять такі характеристики програмного забезпечення:

* цілісність,
* надійність і стійкість,
* продуктивність,
* практичність,
* верифікованість,
* супроводжуваність,
* можливість багаторазового використання,
* мобільність,
* зрозумілість,
* можливість взаємодії,
* ефективність,
* своєчасність реагування,
* видимість процесу розробки.

**Модель якості Дромі**

Модель якості Дромі ґрунтується на виборі критеріїв оцінки. Модель Дромі прагне оцінити якість системи, виходячи з того, що кожен ППмає якість, відмінну від інших.

Модель Дромі допомагає передбачити дефекти ПЗ і вказує на ті властивості ПЗ, нехтування якими може призвести до появи дефектів. Ця модель ґрунтується на відносинах між характеристиками якості та підхарактеристиками, між властивостями ПЗ та характеристиками якості ПЗ.

**Модель якості SATC**

У Центрі забезпечення якості програмного забезпечення NASA (Software Assurance Technology Center, SATC) було розроблено програму метрик, що забезпечує оцінку ризиків проекту, якості продукції та ефективності процесів.

Програма SATC рекомендує окремо відстежувати якість вимог, якість ПЗ, якість тестування та якість виробничих процесів.

Модель якості SATC визначає набір цілей, пов'язаних із програмним продуктом та атрибути процесів відповідно до структури моделі якості програмного забезпечення ISO 9126-1.

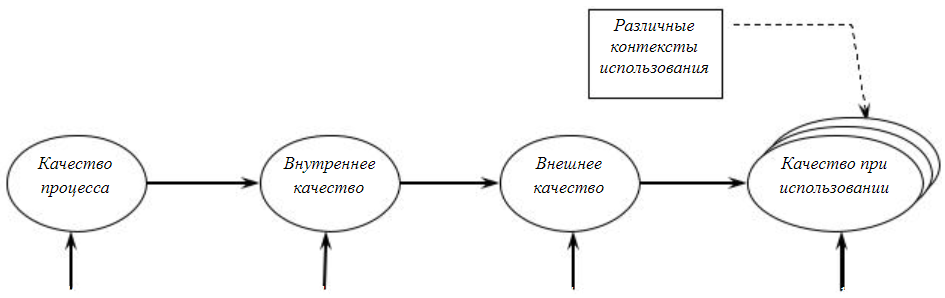
**Модель якості ISO 9126**

Якість програмного забезпечення визначається в стандарті ISO 9126-1 як вся сукупність його характеристик, які стосуються можливості задовольняти висловлені, явно чи очікувані потреби всіх зацікавлених осіб.

Модель якості ISO 9126-1 розрізняє поняття внутрішньої якості, пов'язаного з внутрішньою організацією ПП, без урахування його поведінки, та зовнішньої якості, що характеризує ПЗ з точки зору його поведінки, та проявів ПЗ при використанні в різних контекстах - тієї якості, яка відчувається користувачами за конкретних сценаріїв роботи ПЗ.

Для всіх цих аспектів якості введено метрики, що дозволяють їх оцінити. Крім того, для створення надійного ПЗ суттєво якість технологічних процесів його розробки.

Взаємини між цими аспектами якості за схемою, прийнятою ISO/IEC 9126, показано на Рис 3.



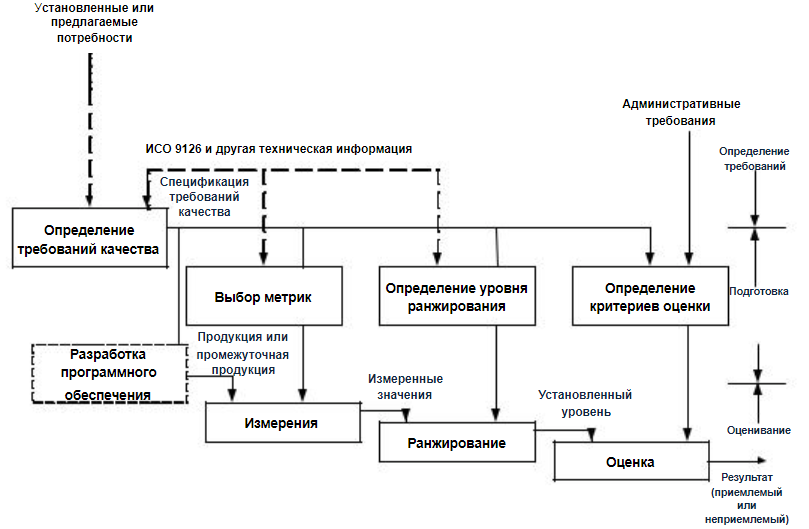
**Рис 3. Основні аспекти якості програмного забезпечення**

На рис. 4 представлені фактори та атрибути зовнішньої та внутрішньої якості програмного забезпечення відповідно до ISO/IEC 9126.



**рис.4. Фактори та атрибути зовнішньої та внутрішньої якості програмного забезпечення**

На рис. 5 наведено модель оцінювання згідно з ISO/IEC 9126.



**Рис 5. Модель оцінювання надійності програмного забезпечення відповідно до стандарту ISO/IEC 9126**

**Модель якості QMOOD**

Джагдіш Банзія і Карл Девіс запропонували ієрархічну модель якості QMOOD для об'єктно-орієнтованого проектування, яка розширює методологію моделі якості Дромі і включає чотири рівні

1. *Визначення показників якості проекту:*  набір атрибутів якості проекту, які використовуються в QMOOD для опису характеристик об'єктно-орієнтованих систем включають: *функціональність, ефективність, зрозумілість, розширюваність, можливість багаторазового використання та гнучкість*.
2. *Визначення об'єктно-орієнтованих властивостей проекту:*  властивості проекту можуть бути визначені у процесі дослідження внутрішньої та зовнішньої структури, функціональності компонентів проекту, атрибутів, методів і класів.

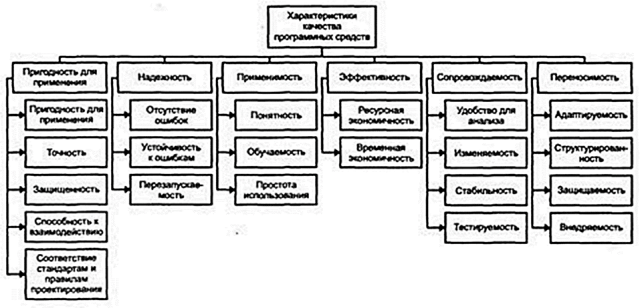
Структурною та об'єктно-орієнтованою множиною властивостей проекту, які використовуються в QMOOD, є*: розмір проекту, ієрархічна структура, інкапсуляція, пов'язаність, склад проекту, успадкування, поліморфізм, обмін інформацією, складність*.

1. *Визначення об'єктно-орієнтованих метриків проекту:*  різні об'єктно-орієнтовані метрики проекту.
2. *Визначення об'єктно-орієнтованих властивостей проекту* : для визначення архітектури об’ектно-орієнтовного проекту введено компоненти проєкту і низку нових об'єктноорієнтованих метрик.

**Багаторівневий підхід до моделей якості.**

На рис 6 подано характеристики якості програмних засобів ([ГОСТ 28195-89](http://docs.nevacert.ru/files/gost/gost_28195-1989.pdfhttp:/docs.nevacert.ru/files/gost/gost_28195-1989.pdf) голос із минулого для довідки).

В основі моделі якості лежить багаторівневий підхід (кількість шарів може бути 2 (моделі МакКола і Боема) або 3 (те ж саме з додаванням метрики)).



**Рис 6. Характеристики якості програмних засобів (за ГОСТ 28195-89).**

Порівняльний аналіз характеристик різних моделей якості ПЗ наведено у таблиці 2.1.

**Таблиця 2.1. Порівняльний аналіз моделей якості програмного забезпечення**

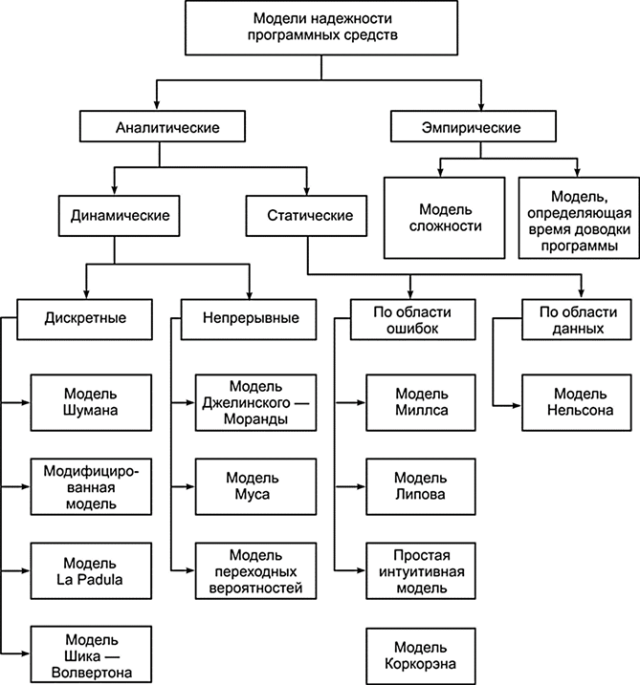
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Характеристики якості | Маккол | Боем | FURPS/ FURPS+ | Гецці | Дромі | ISO 9126 | Казман | Хосраві | Шармоа |
| Коректність |  |  |  | + |  |  |  |  |  |
| Надійність | + | + | + | + | + | + | + |  | + |
| Коректність | + |  |  |  |  |  |  |  |  |
| Ефективність | + | + | + | + | + | + | + |  | + |
| Гнучкість | + |  |  | + |  |  | + | + |  |
| Функціональність |  |  | + |  | + | + | + |  | + |
| Ергономічність проетування |  | + |  |  |  |  |  |  |  |
| Цілісність |  |  |  | + |  |  |  |  |  |
| Функціональна сумісність | + |  |  |  |  |  |  |  |  |
| Супроводжуваність | + | + | + | + | + | + | + |  | + |
| Модифікованість |  | + |  |  |  |  |  |  |  |
| Продуктивність |  |  | + |  |  |  |  |  |  |
| Мобільність | + | + |  | + | + | + |  |  | + |
| Зрілість процесу |  |  |  |  | + |  |  |  |  |
| Можливість багаторазового використання | + |  |  | + |  |  |  | + |  |
| Стійкість |  |  |  |  |  |  |  | + |  |
| Масштабованість |  |  |  |  |  |  |  | + |  |
| Безпека |  |  | + |  |  |  | + |  |  |
| Експлуатаційна придатність |  |  | + |  |  |  |  |  |  |
| Тестованість | + | + |  |  |  |  | + |  |  |
| Зрозумілість |  | + | + |  |  |  |  |  |  |
| Практичність | + |  | + | + | + | + | + | + | + |

Моделі оцінки надійності

**Класифікація моделей**

Термін  *модель надійності програмного забезпечення* зазвичай відноситься до математичної моделі, побудованої для оцінки залежності надійності програмного забезпечення від деяких параметрів.

|  |
| --- |
| Рисунок 7. Класифікаційна схема моделей надійності програмних засобів |

****

**Вимоги до технології та засобів автоматизації розробки складних програмних засобів**

Для забезпечення якості та надійності ПС в практиці розробки стандартами  *рекомендується формулювати наступні вимоги:*

* до об'єкта розробки на поточному етапі - до його програмних та інформаційних компонентів, а також до інтерфейсу між ними та зовнішнім середовищем;
* до процесу, технології та організації виконання сукупності робіт та документів кожного етапу;
* до методів та характеристик засобів автоматизації виконання робіт, що забезпечують необхідну надійність функціонування та якість ПС;
* до методів та засобів контролю, вимірювання та документування якості процесів та результатів виконаних робіт.

Вимоги до інструментальних засобів автоматизації розробки надійних ПС найповніше викладені у стандарті  ***IEEE 1209-1992.***

**Модель Шумана**

Модель Шумана будується на основі кількох критеріїв:

* загальна кількість команд у програмі машинною мовою постійна;
* на початку компонувальних випробувань число помилок дорівнює деякій постійній величині, і в міру виправлення помилок їх стає менше. Під час випробувань програми нових помилок не вноситься;
* помилки спочатку помітні, по сумарному числу виправлених помилок можна будувати висновки про ті, що залишилися;
* інтенсивність відмов програми пропорційна числу залишкових помилок.

Передбачається, що на початок тестування (тобто. у момент t=0) є M помилок. Протягом часу тестування виявляється ε 1 (t) помилок у розрахунку на одну команду в машинній мові.

Тоді питома кількість помилок на одну машинну команду, що залишилися в системі після часу тестування, дорівнює:

https://i0.wp.com/pdnr.ru/infopediasu/baza23/15107609623304.files/image018.png?w=640&ssl=1 (1)

де I - загальна кількість машинних команд, що передбачається постійним у рамках етапу тестування.

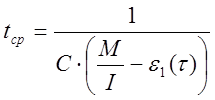
Передбачається, що значення функції кількості помилок Z(t) пропорційно кількості помилок, що залишилися в програмі після витраченого на тестування часу τ.

Z (t) = C \* ε 2 (τ),

де С - деяка стала, t - час роботи програми без відмов.

Тоді, якщо час t безвідмовної роботи програми відраховується від точки t = 0, і залишається фіксованим, функція надійності, або ймовірність безвідмовної роботи на інтервалі від 0 до t, дорівнює

https://i0.wp.com/pdnr.ru/infopediasu/baza23/15107609623304.files/image020.png?w=640&ssl=1 (2)

 (3)

Нам необхідно знайти початкове значення помилок M і коефіцієнт пропорційності С. Ці невідомі оцінюються шляхом пропуску функціонального тесту у двох точках змінної осі налагодження t a  і t в обраних так, що ε 1 (t a )<ε 1 (t d ).

У процесі тестування збирається інформація про час і кількість помилок кожному прогоні, тобто. загальний час тестування складається з часу кожного прогону:

τ = 1  + 2  + 3  + … + n .

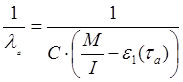
Припускаючи, що інтенсивність появи помилок стала і дорівнює λ, можна обчислити її як число помилок в одиницю часу,

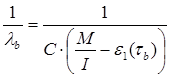
 (4)

де A i - кількість помилок на i - му прогоні. Тоді

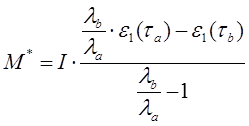
 .(5)

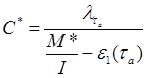
Маючи дані для двох різних моментів тестування t a  і t в можна порівняти рівняння (3) при τ a і τ b :

 (6)

 (7)

Зі співвідношень (6) і (7) знайдемо невідомий параметр С і М:

 (8)

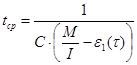
 (9)

Отримавши невідомі M \*  і C \* можна розрахувати надійність програми за формулою (2).

**приклад 1.**

Програма містить 2 000 командних рядків, з них до початку експлуатації (після періоду налагодження) 15 командних рядків містять помилки. Після 20 днів роботи виявлено 1 помилку. Знайти середній час безпомилкової роботи програми та інтенсивність відмов програми при коефіцієнті пропорційності, що дорівнює 0,7.

|  |  |
| --- | --- |
| I= | 2000 |
| M= | 15 |
| t= | 20 |
| x= | 1 |
| C= | 0,7 |
| https://i0.wp.com/pdnr.ru/infopediasu/baza23/15107609623304.files/image036.png?w=640&ssl=1 |  |
| https://i0.wp.com/pdnr.ru/infopediasu/baza23/15107609623304.files/image038.png?w=640&ssl=1 |  |



|  |  |  |  |
| --- | --- | --- | --- |
| E1(t)= | 0,0005 |  |  |
| E2(t)= | 0,007 |  |  |
| P(t)= | 0,906649 |  |  |
| tср = | 204,0816 |  |  |
| λ= | 0,0049 | - Інтенсивність відмов |  |

**приклад 2.**

На умовах прикладу 1 визначити можливість безпомилкової роботи програми протягом 90 діб.

|  |  |
| --- | --- |
| I= | 2000 |
| M= | 15 |
| t= | 90 |
| x= | 1 |
| C= | 0,7 |
| P(t)=0,643393 |  |

**приклад 3.**

Визначити початкову кількість можливих помилок у програмі, що містить 2 000 командних рядків, якщо протягом перших 60 діб експлуатації було виявлено 2 помилки, а за наступні 40 діб було виявлено одну помилку. Визначити T 0  – середній час безпомилкової роботи, що відповідає першому та другому періоду експлуатації програми та коефіцієнт пропорційності.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| I= | 2000 |  |  |  |
| t1= | 60 діб |  |  |  |
| t2= | 100 діб |  |  |  |
| x1= | 2 помилок |  |  |  |
| x2= | 3 помилок |  |  |  |
| T0= | 30,333333 |  |  |  |
| λ1= 0,033333 |  |  |  |  |
| λ2= | 0,03 |  |  |  |
| C= | 6,666667 |  |  |  |
| E1(t1)= | 0,001 |  |  |  |
| E2(t2)= | 0,0015 |  |  |  |
| M= | 12 |  |  |  |
| Л2/Л1= | 0,9 |  |  |  |
|  |  |  |  |  |

**Модель Міллса**

Використання цієї моделі передбачає необхідність перед початком тестування штучно засмічувати програму, тобто вносити в неї кілька відомих помилок. Помилки вносяться випадково і фіксуються в протоколі штучних помилок. Фахівець, який проводить тестування, не знає ні кількості, ні характеру внесених помилок до моменту оцінки показників надійності моделі Міллса.

Передбачається, що всі помилки (як природні, так і штучно внесені) мають рівну ймовірність бути знайденими в процесі тестування.

Тестуючи програму протягом деякого часу, збирають статистику про помилки. У момент оцінки надійності за протоколом штучних помилок усі помилки поділяються на власні та штучні. Співвідношення, зване формулою Міллса,

N = https://i0.wp.com/pdnr.ru/infopediasu/baza23/15107609623304.files/image042.png?w=640&ssl=1

Дає можливість оцінити початкове число помилок у програмі N. Тут S – кількість штучно внесених помилок; n – кількість знайдених помилок; V – число виявлених на момент оцінки штучних помилок.

**Модель Джелінскі-Моранди**

Основне положення, на якому базується модель, полягає в тому, що в процесі тестування значення інтервалів часу тестування між виявленням двох помилок має експоненційний розподіл з інтенсивністю відмов, пропорційною числу ще не виявлених помилок.

Кожна виявлена ​​помилка усувається, кількість помилок, що залишилися, зменшується на одиницю.

Функція щільності розподілу часу виявлення i-ї помилки, що відраховується від моменту виявлення (i – 1)-ї помилки, має вигляд

https://i0.wp.com/pdnr.ru/infopediasu/baza23/15107609623304.files/image044.png?w=640&ssl=1 )

Де  https://i0.wp.com/pdnr.ru/infopediasu/baza23/15107609623304.files/image046.png?w=640&ssl=1  – інтенсивність відмов, яка пропорційна числу ще не виявлених помилок у програмі:

https://i0.wp.com/pdnr.ru/infopediasu/baza23/15107609623304.files/image048.png?w=640&ssl=1

Де N – число помилок, що спочатку присутні у програмі;

С – коефіцієнт пропорційності.

Найбільш ймовірні значення величин N і визначаються на основі даних, отриманих при тестуванні.

І тому фіксують час виконання програми до чергового відмови t1,t2,t3,…,tk. Значення N і С можна отримати, розв'язавши систему рівнянь

https://i0.wp.com/pdnr.ru/infopediasu/baza23/15107609623304.files/image050.png?w=640&ssl=1

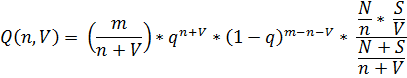
Де

Q =  https://i0.wp.com/pdnr.ru/infopediasu/baza23/15107609623304.files/image052.png?w=640&ssl=1 ; A =  https://i0.wp.com/pdnr.ru/infopediasu/baza23/15107609623304.files/image054.png?w=640&ssl=1 ; B =  https://i0.wp.com/pdnr.ru/infopediasu/baza23/15107609623304.files/image056.png?w=640&ssl=1 .

Щоб отримати числові значення λ, потрібно підставити замість N і їх можливі значення N і C. Розрахувавши До значень за формулою (5) і підставивши їх у вираз (4), можна визначити можливість безвідмовної роботи на різних часових інтервалах.

**Модель Липова**

Ліпов модифікував модель Міллса, розглянувши можливість виявлення помилки при використанні різного числа тестів. Якщо зробити те ж припущення, що і в моделі Міллса, тобто власні та штучні помилки мають рівну ймовірність бути знайденими, то ймовірність виявлення n власних і V внесених помилок



Де m – кількість використовуваних тестів, q – ймовірність виявлення помилки у кожному з m тестів, розрахована за такою формулою  https://i0.wp.com/pdnr.ru/infopediasu/baza23/15107609623304.files/image060.png?w=640&ssl=1 ;

S – загальна кількість штучно внесених помилок;

N – кількість власних помилок, що у ПЗ до початку тестування.

Метрики якості програм

Для вимірювання характеристик якості використовують метрики.

**Означення.**

Метрика – це міра, що дозволяє отримати чисельне значення деякої властивості ПЗ або його специфікацій, для того, щоб розуміти в якій мірі система, її компоненти або процес відповідають встановленим нормам

Метрики повинні

* чисельно характеризувати основну цільову функцію програми;
* забезпечувати можливість визначення витрат, необхідних досягнення необхідного рівня якості, і навіть ступеня впливу на показник якості різних зовнішніх чинників;
* бути по можливості простим, добре вимірним та мати малу дисперсію.

Метрики якості програм – система вимірювання якості програм. Ці виміри можуть проводитися на рівні критеріїв якиості програм або на рівні окремих характеристик якості.

У першому випадку система вимірів дозволяє безпосередньо порівнювати програми з якості. При цьому самі виміри не можуть бути без суб'єктивних оцінок властивостей програм.

У другому випадку вимірювання характеристик можна виконати об'єктивно та достовірно, але оцінка якості ПЗ у цілому буде пов'язана з суб'єктивною інтерпретацією одержуваних оцінок.

**Метричні шкали**

Залежно від характеристик та особливостей застосовуваних метрик їм ставляться у відповідність різні вимірювальні шкали.

Номінальній шкалою відповідають метрики, що класифікують програми на типи за ознакою наявності чи відсутності деякої характеристики без урахування градацій.

Порядковій шкалі відповідають метрики, що дозволяють ранжувати деякі характеристики шляхом порівняння з опорними значеннями, тобто. вимір за цією шкалою фактично визначає взаємне становище конкретних програм.

Інтервальної шкалою відповідають метрики, які показують не тільки відносне становище програм, а й те, як далеко вони відстоять одна від одної.

Відносній шкалі відповідають метрики, що дозволяють не тільки розмістити програми певним чином і оцінити їх положення відносно один одного, але й визначити, як далеко оцінки відстоять від кордону, починаючи з якого характеристика може бути виміряна.

**Метрики складності**

В залежності від кваліфікації і бажання програміста (якщо оцінка продуктивності його роботи буде здійснюватись за об’ємом коду в одиницю часу) одна і та ж задача в різноиу виконанні буде мати програми різної довжини.

Де та отимальна довжина, за межами якої буде збитковий код, а значить і знижена його якість ?

При оцінці складності програм, як правило, виділяють три основні групи метрик:

* метрики розміру програм
* метрики  складності потоку управління програм
* метрики  складності потоку даних програм

Традиційною характеристикою розміру програм є кількість рядків вихідного тексту. Під рядком розуміється будь-який оператор програми, оскільки саме оператор, а не окремий рядок є тим інтелектуальним «квантом» програми, спираючись на який можна будувати метрики складності її створення.

Безпосередній вимір розміру програми, незважаючи на свою простоту, дає добрі результати.

Звичайно, оцінка розміру програми недостатня для ухвалення рішення про її складність, але цілком застосовна для класифікації програм, що істотно різняться обсягами.

При зменшенні відмінностей обсягом програм першому плані висуваються оцінки інших чинників, надають впливом геть складність.

Таким чином, оцінка розміру програми є оцінкою за номінальною шкалою, на основі якої визначаються лише категорії програм без уточнення оцінки для кожної категорії.

До групи оцінок розміру програм можна віднести також метрику Холстеда.

**Метрика Холстеда** Метрика Холстеда відноситься до метрик, що обчислюються на підставі аналізу числа рядків та синтаксичних елементів вихідного коду програми.

Основу метрики Холстеда становлять чотири вимірювані характеристики програми:

* · NUOprtr (Number of Unique Operators) - кількість унікальних операторів програми, включаючи символи-розділювачі, імена процедур та знаки операцій (словник операторів);
* · NUOprnd (Number of Unique Operands) - число унікальних операндів програми (словник операндів);
* · Noprtr (Number of Operators) - загальна кількість операторів у програмі;
* · Noprnd (Number of Operands) - загальна кількість операндів у програмі.

Метрики Холстеда відбивають лексичний підхід до вимірювання характеристик програмного забезпечення, заснований на властивостях вимірних алгоритмів. Властивості будь-якого опису алгоритму (або програми для ЕОМ), на думку Холстеда, можуть бути виміряні або обчислені на основі наступних метричних характеристик (оцінних елементів):

* n1 – кількість різних операторів програми;
* n2 - кількість різних операндів програми;
* N1 – загальна кількість операторів програми;
* N2 – загальна кількість операндів програми.

На їх основі Холстед визначає такі метрики:

*Словник програми (в умовних одиницях)*

n = n1+n2, (1)

*Теоретичний словник програми*, тобто. словниковий запас, необхідний написання програми, з огляду на те, що необхідна функція вже реалізована у цій мові і, отже, програма зводиться до виклику цієї функції.

n \* = n1 \* + n2 \*

*Довжина програми* (в умовних одиницях)

N = N1+N2 (2)

*Теоретична довжина програми* (в умовних одиницях)

С = (n1 log 2 n1) + (n2log 2 n2), (3)

Вводячи цю оцінку, Холстед виходить із основних концепцій теорії інформації, за аналогією з якими частота використання операторів та операндів у програмі пропорційна двійковому логарифму кількості їх типів.

Таким чином, вираз (3) є ідеалізованою апроксимацією (2), тобто справедливо для потенційно коректних програм, вільних від надмірності або недосконалостей (стилістичних помилок).

Недосконалостями можна вважати такі ситуації:

* o наступна операція знищує результати попередньої без їх використання;
* o присутні тотожні вирази, які вирішують однакові завдання;
* o однієї і тієї ж змінної призначаються різні імена тощо.

Подібні ситуації спричиняють зміну N без зміни n.

Холстед стверджує, що для стилістично коректних програм відхилення в оцінці теоретичної довжини від реальної N не перевищує 10%.

Пропонується використовувати як еталонне значення довжини програми зі словником n. Довжина коректно складеної програми N, тобто програми, вільної від надмірності і має словник n, не повинна відхилятися від теоретичної довжини програми більш ніж на 10%. Таким чином, вимірюючи n1, n2, N1 і N2 і зіставляючи значення N і С для деякої програми, при більш ніж 10% відхиленні можна говорити про наявність у програмі стилістичних помилок, тобто недосконалостей.

Насправді N і З найчастіше істотно різняться.

*Об'єм програми* (у бітах, визначення обсягу програми передбачає, що позначення операндів і операторів вимагає не більше одного символу).

V = N log 2 (n) (4)

*Потенційний обсяг програми* (вважаючи, що в програмах, ідеальних з точки зору економії витрат пам'яті,

по-перше: оператори та операнди не повторюються,

по-друге: всі операнди є або вхідними, або вихідними параметрами і

по-третє: для запису тексту програми достатньо двох операторів (описи заголовка процедури-функції та надання значення))

V \* = n \* log 2 n \* (5)

де n2\* - загальна кількість вхідних та вихідних параметрів.

· Рівень якості програми (в умовних одиницях, якщо n2 \* = 2)

L = V \* / V (6)

Базою для введення цієї характеристики є припущення, що при зниженні стилістичної якості програмування зменшується змістовне навантаження на кожен компонент програми і, як наслідок, розширюється обсяг реалізації вихідного алгоритму.

*Наступна група оцінок складності програм — це метрики складності потоку управління програм.*

Типовими представниками цієї групи є метрика Маккейба. Зазвичай, з її допомогою оперують або щільністю управляючих переходів усередині програм, або взаємозв'язками цих переходів. І в тому, і в іншому випадку стало традиційним представлення програм у вигляді керуючого орієнтованого графа

https://studme.org/htm/img/15/4913/24.png

де *V* - вершини, що відповідають операторам; *Е* – дуги, що відповідають переходам.

**Метрика Маккейба**

У основі лежить ідея оцінки складності ПЗ за кількістю базисних шляхів у керуючому графі, тобто. таких шляхів, компонуючи які можна отримати всілякі шляхи із входу графа у виходи.

Основною метрикою складності МакКейб пропонує вважати цикломатичну складність графа програми, або, як її ще називають, *цикломатичне число МакКейба, що* характеризує трудомісткість тестування програми.

Програми (алгоритм, специфікація) має бути представлена ​​у вигляді управляючого орієнтованого графа https://studfile.net/html/14845/114/html_yKBB5PZpD3.8EZ3/img-qRSyan.png з *V*вершинами та *E*дугами, де вершини відповідають операторам, а дуги – переходам від одного оператора до іншого.

Граф, що описує програму у виглялі вершин-операторів і дуг-переходів , називають графом управління чи управляючим графом програми .

Зазвичай враховують тільки здійсненні оператори , крім операторів опису даних.

Лінійні ділянки програми можна замінити одним вузлом графа .

Бажано перетворити оператори циклу на еквівалентну послідовність операторів розгалуження , додати оператори підсумовування (лічильники) числа повторень циклу з «верхнім» або «нижнім» закінченням.

Цикломатичничне число графа управління

https://studfile.net/html/14845/114/html_yKBB5PZpD3.8EZ3/img-h4l322.png,

m – кількість ребер графа;

n – кількість вершин графа.

Величину M називають цикломатичним числом Маккейба .

Цикломатична складність програми – структурна (топологічна) міра складності програм, що використовується для вимірювання якості програмного забезпечення, заснована на методах статичного аналізу коду.

Цикломатична складність програми дорівнює збільшеному на одиницю цикломатичному числу графа програми.

Для обчислення цикломатичної складності використовується граф потоку управління програми : вузли графа відповідають неподільним групам команд програми та орієнтованим ребрам, кожен з яких з'єднує два вузли та відповідає двом командам, друга з яких може бути виконана відразу після першої.

Ця стратегія тестування називається основним маршрутом тестування Маккейба : тестування кожного лінійного незалежного маршруту через програму – у цьому випадку кількість тестів має дорівнювати циклічному складності програми.

Цикломатична складність частини програмного коду – кількість лінійно незалежних маршрутів через програмний код.

|  |  |
| --- | --- |
| Вихідний код не містить жодних точок рішень (IF, FOR) | Складність = 1 (є лише один маршрут) |
| Код має єдиний оператор IF, що містить просту умову | Два шляхи: один – через IF як TRUE та один – як FALSE |

Теоретично графів цикломатичне число орієнтованого графа

https://studfile.net/html/14845/114/html_yKBB5PZpD3.8EZ3/img-0pp0O6.png

m – кількість ребер;

n – кількість вершин;

p – кількість компонентів зв'язності графа.

Число компонентів зв'язності p можна розглядати як мінімально необхідну кількість ребер , які потрібно додати до графа, щоб зробити його повнозв'язковим .

Справедливо вважати, що з будь-якого графа управління програми число компонентів зв'язності дорівнює одиниці ( p = 1) .

Підстановка р = 1 у формулу визначення цикломатичного числа дає цикломатичне число Маккейба – визначає кількість незалежних контурів у повнозв'язному графі та, як наслідок, кількість різних шляхів, що ведуть з початкової вершини до кінцевої .

При оцінці складності програми з використанням цикломатичного числа Маккейба діє правило: якщо цикломатичне число більше за Z > 10, програма має зайву складність і її слід розбити на складові частини з меншим значенням цикломатичного числа.

**Метрика Майєрса** є розширення метрики Маккейба. Суть підходу Г. Майєрса полягає у наступному. Мірою складності запропоновано інтервал

[v, v2 ],

де v, - цикломатична міра, v 2 - кількість окремих умов плюс одиниця. При цьому оператор DO приймається як одна умова, а оператор CASE з результатами приймається за (п - 1) умов *.*Введений міра отримала назву *інтервальної міри.*

**Метрика Джилба**

Однією з найпростіших, але, як свідчить практика, досить ефективних оцінок складності програм є метрика Т. Джилба, у якій логічна складність програми визначається насиченістю програми виразами типу IF-THEN-ELSE.

При цьому вводяться дві характеристики:

CL - абсолютна складність програми, що характеризується кількістю операторів умови;

cl - відносна складність програми, що характеризується насиченістю програми операторами умови, тобто cl визначається як відношення CL до загального числа операторів.

**З чого розпочати забезпечення якості?**

Організація забезпечення якості починається з наступних кроків

* Узгодження загальних моделей
* Визначення послідовність дій з забезпечення якості
* Впровадження стандартів та процесів
* Аналіз завершених проектів
* Аналіз та навчання на основі даних про дефекти
* Використання власного досвіду.

Вони відносно прості в реалізації та приносять реальну відчутну користь:

**Забезпечення якості - це процес навчання** :

* вивчення того, що працює неправильно, і як це виправити;
* вивчення того, що працює правильно, та за яких обставин та
* вивчення того, як краще виконувати свою роботу з кожним новим проектом.

Будь-яка організація, що займається забезпеченням якості, постійно навчається. Перший крок — зробити забезпечення якості невід'ємною частиною розробки продукції.

Питання для самоконтролю

1. Дайте визначення помилки та дефекту.
2. Як змінюється ціна помилки по мірі просування процесу розробки ПЗ в часі ?
3. В яких умовах тестування програм може вплинути на якість ПП ?
4. Що впливає на якість ПП ?
5. В чому різниця між внутрішньою та зовнішньою якостями ПП ?
6. Для чого забезпечувати внутрішню якість ?
7. Для чого потрібні відступи ?
8. Що таке дисципліна відступів ?
9. Які проблеми з відступами можуть бути при зміні редактора коду ?
10. Як можна скористатись клавішею прогалини (Space) для поліпшення сприйняття коду ?
11. Для чого потрібні правила іменування змінних, класів і функцій в коді ?
12. Як правильно іменувати змінні та інше ?
13. В чому суть угорської нотації ?
14. Що тааке самодокументовність коду ?
15. Які рекомендації щодо вживання коментарів ?
16. Які рекомедації з використання констант ?
17. Якого розміру програмних блоків бажано дотримуватись ?
18. Чи існують стандарти стилю оформлення коду ?
19. Який код називають лінійним ?
20. Що порушує лінійність коду ?
21. Що таке надійність програми ?
22. Охарактеризуйте зручність використання ПП
23. Як ви поясните ефективність ПП ?
24. Що сприяє ремонтопридатності ПП ?
25. В чому полягає стійкість до збоїв ?
26. Що таке кросплатформність ПП?
27. Що таке експлуатаційна придатність ?
28. Що підрозумівається пвд продуктивністю ПП ?
29. Для чого потрібні моделі якості ?
30. Які типи моделей якості ви знаєте ?
31. Які категорії моделей вам відомі ?
32. Що відноситься до функціональних характеристик ?
33. Нааведіть приклади нефункціональних характеристик
34. Що що являє собою модель надійності ?
35. Які класи моделей надійності вам відомі ?
36. Що таке складність програми ?
37. Що наазивають метрикою якості ПП ?
38. В чому ідея метрик Маккейба ?
39. Перерахуйте метрики складності
40. Для чого служать метрики Холстеда ?
41. На чому грунтується метрика Джилба ?
42. Перерахуйте основні кроки впровадження системи забезпечення якості