МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ

ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ

ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Опорний конспект лекцій
з курсу

"Якість програмного забезпечення та тестування"

для студентів напрямку підготовки "Програмна інженерія"

Тернопіль – 2012

Козак О.Л. Опорний конспект лекцій з курсу “Якість програмного забезпечення та тестування” для студентів напрямку підготовки “Програмна інженерія” / О.Л. Козак. – Тернопіль, 2012. – с.72

Анотація. У навчальному посібнику висвітлено ряд тем, вивчення яких дозволяє сформування у студента систему теоретичних і практичних знань з оцінки якості та тестування програмного забезпечення. В курсі розглянуті теми, які актуальні для фахівців на шляху до створення працездатного і якісного програмного продукту. Метою курсу є засвоєння методів є вивчення сучасних парадигм та технологій забезпечення якості програмного забезпечення при його розробці, методів тестування, верифікації та атестації на основі стандартів якості. Викладений матеріал повинен сприяти формуванню висококваліфікованих фахівців у галузі програмного забезпечення.

Укладач: **Козак Олександра Леонідівна**, к.т.н., доцент кафедри комп’ютерних наук ТНЕУ.

Відповідальний за випуск: **Дивак Микола Петрович**, д.т.н., професор, завідувач кафедри комп’ютерних наук ТНЕУ

Затверджено на засіданні кафедри комп’ютерних наук ТНЕУ.
Протокол № 15 від 18.05.2012 р.

ЗМІСТ

ТЕМА 1. ОСНОВИ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

1.1 Якість та місце тестування у життєвому циклі програмних продуктів

За десятиліття досвіду побудови програмних систем був напрацьований ряд типових схем послідовності виконання робіт при проектуванні та розробки ПС. Такі схеми отримали назву моделей ЖЦ.

Модель життєвого циклу - це схема виконання робіт та завдань у рамках процесів, що забезпечують розробку, експлуатацію та супровід програмного продукту, і відбиває еволюцію ПС, починаючи від формулювання вимог до неї до припинення користуватися нею.

Історично в цю схему робіт включають:

- розробку вимог або технічного завдання;
- розробку системи або технічного проекту;
- програмування або робоче проектування;
- пробну експлуатацію;
- супровід та поліпшення;
- зняття з експлуатації.

Основне призначення моделей ЖЦ полягає в наступному:

- планування і розподіл робіт між розробниками і ресурсів, а також управління програмним проектом;
- забезпечення взаємодії між розробниками проекту та замовником;
- спостереження і контроль робіт, оцінювання проміжних продуктів ЖЦ на дотримання специфікацій вимог, правильне їх виконання, оцінювання продукту та реальних витрат, у тому числі і по застосовуваних програмних засобів та інструментів;
- узгодження проміжних результатів із замовником;
- перевірка правильності кінцевого продукту шляхом його тестування на запланованих і узгоджених з замовником наборах тестів;
- оцінювання відповідності характеристик якості отриманого продукту заданим вимогам;
- обговорення використовуваних процесів ЖЦ в плані оцінки їх можливостей і недоліків, які виявили при їх застосуванні, а також визначення напрямків удосконалення або модернізації ЖЦ та ін.

У зв'язку з такими завданнями, що покладаються на модель ЖЦ, необхідно зробити правильний вибір процесів, їх завдань та дій для побудови моделі ЖЦ ПС, яка задовольняє концептуальній ідеї проектованої системи з урахуванням її складності та масштабу робіт. У модель ЖЦ обов'язково включаються процеси реалізації робіт і завдань, що забезпечують створення проміжного продукту і перехід до наступного процесу моделі.

Процеси ЖЦ стандарту ISO / IEC 12207

При виборі схеми моделі ЖЦ для конкретної предметної області, вирішуються питання включення важливих для створюваного продукту видів робіт або не включення несуттєвих робіт. На сьогодні основою формування нової моделі ЖЦ для конкретної прикладної системи є стандарт ISO / IEC 12207, що задає повний набір процесів (понад 40), що охоплює всі можливі види робіт і завдань, пов'язаних з побудовою ПС, починаючи з аналізу предметної області і закінчуючи виготовленням відповідного продукту. Цей стандарт містить основні та допоміжні процеси (рис. 1.1 та рис. 1.2).

Рис. 1.1. Схема основних процесів ЖЦ ПС

На рис. 1.1. представлені процеси, пов'язані безпосередньо з розробкою ПС. До категорії основних процесів відносяться також "первинні" процеси, що визначають порядок підготовки договору на розробку ПС, моніторинг діяльності постачальників ПС замовнику.

Стандарт ISO / IEC 12207 надає структуру процесів ЖЦ, але не зобов'язує використовувати всі процеси в моделі ЖЦ ПЗ або в конкретній методології розробки ПЗ

Рис. 1.2. Схема допоміжних процесів ЖЦ ПЗ

Будучи стандартом високого рівня, він не задає деталі того, як треба виконувати дії або завдання, складові процеси. Він також не ставить вимог до формату і змісту документів, що випускаються на різних процесах.

Процеси, дії і завдання наведені в стандарті в найбільш загальній природній послідовності. Це не означає, що в такій же послідовності вони повинні бути застосовані в конкретній моделі ЖЦ ПС. В залежності від проекту процеси, дії і завдання стандарту вибираються, упорядковуються і включаються в модель ЖЦ. При застосуванні вони можуть перекривати, переривати один одного, виконуватися ітераційно або рекурсивно. Це визначає "динамічний" характер стандарту і дозволяє реалізувати з його допомогою довільну модель ЖЦ ПС. Тому організації, що має намір застосувати цей стандарт у своїй роботі, знадобляться додаткові стандарти чи процедури, що визначають різні деталі по

застосуванню вибраних елементів ЖЦ. Відзначимо, що комітет ISO випускає керівництва і процедури, що доповнюють стандарт 12207.

Крім цього, стандарт ISO / IEC 12207 надає основу для прийняття ряду інших пов'язаних з ним стандартів, таких як стандарти з управління ПЗ, забезпечення якості, верифікації та валідації, управління конфігурацією, метрики ПЗ і т.д.

З цього стандарту можна вибрати лише ті процеси, які найбільше підходять для реалізації конкретної ПС. Обов'язковими є основні процеси, які присутні у всіх відомих моделях ЖЦ. В залежності від цілей і завдань предметної області вони можуть бути поповнені додатковими (документування, забезпечення якості, верифікація та валідація тощо) та організаційними (планування, управління та ін) процесами цього стандарту. Розробник приймає рішення про включення в нову створювану модель ЖЦ процесу забезпечення якості компонентів і системи управління проектом або визначення набору перевірочних (верифікаціонних) процедур для забезпечення правильності продукту і відповідності його заданим вимогам.

Процеси, включені в модель ЖЦ, призначені для реалізації стандартних задач процесів ЖЦ і можуть залучати інші процеси для реалізації спеціалізованих завдань системи (наприклад, захисту даних). Інтерфейси (входи і виходи) будь-яких двох процесів ЖЦ повинні бути мінімальними і кожен з них повинен задовольняти наступним правилами:

• якщо процес А викликається процесом В і тільки процесом В, то А належить В;

• якщо функція викликається більш ніж одним процесом, то вона стає окремим процесом;

• перевірка будь-якої функції в ЖЦ є обов'язковою.

Іншими словами, якщо завдання потрібне більше ніж одному процесу, то вона може стати процесом, використовуваним однократно або багаторазово протягом життя конкретної системи. Кожен процес повинен мати внутрішню структуру, встановлену у відповідності з тим, що він повинен виконувати.

Процеси моделі ЖЦ орієнтовані на розробника системи. Він може виконувати один або декілька процесів і процес може бути виконаний одним або кількома розробниками. При цьому один з них є відповідальним за один процес або за всі процеси моделі, навіть якщо окремі роботи виконує інший розробник.

Створювана модель ЖЦ узгоджується з конкретними методиками розробки систем та відповідними стандартами в області програмної інженерії або розробляються самостійно для проекту з урахуванням його можливостей і особливостей. Іншими словами, кожен процес ЖЦ підкріплюється обраними для реалізації завдань ПС засобами, методами програмування і методикою їх застосування та виконання.

Важливу роль при формуванні моделі ЖЦ мають організаційні аспекти:

• планування послідовності робіт та строків їх виконання;

• підбір і підготовка ресурсів (людських, програмних і технічних) для виконання робіт;

• оцінка можливостей реалізації проекту в задані терміни, вартість і ресурси.

Впровадження моделі ЖЦ в практичну діяльність по створенню програмного продукту дозволяє впорядкувати взаємини між суб'єктами процесу розробки ПС і враховувати динаміку модифікації вимог до проекту і до системи.

Типи моделей ЖЦ

Розглянуті питання стали джерелом формування різних видів моделей ЖЦ на основі процесного підходу до розробки програмних проектів. До широко використовуваним типам моделей ЖЦ відносяться наступні: каскадна, спірально, інкрементна, еволюційна, стандартизована та ін.

Каскадна модель ЖЦ

Однією з перших стала застосовуватися каскадна модель, в якій кожна робота виконується один раз і в тому порядку, як це представлено у моделі (мал. 2.4).

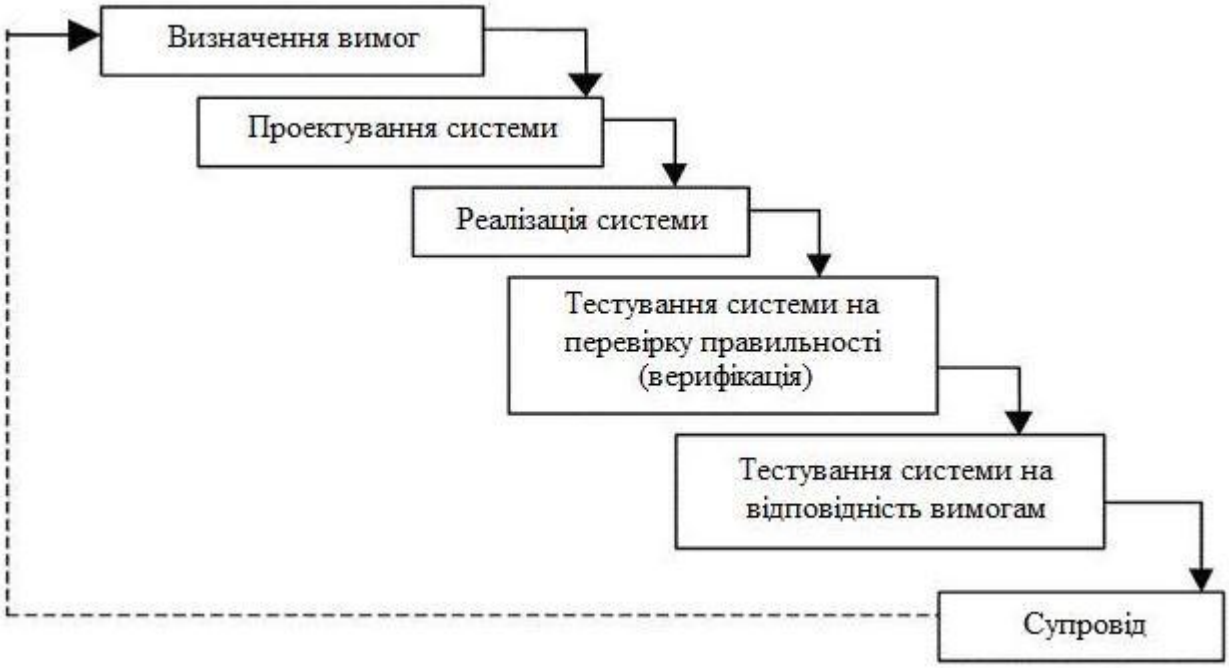

Рис. 1.4. Каскадна модель ЖЦ програмних систем

Тобто робиться припущення, що кожна робота буде виконана настільки ретельно, що після її завершення і переходу до наступного етапу повернення до попереднього не буде потрібно.

Розробник перевіряє проміжний результат різними відомими методами верифікації та фіксує його в якості готового еталону для наступного процесу.

Відповідно до цієї моделі ЖЦ роботи та завдання процесу розробки зазвичай виконуються послідовно, як це представлено у схемі. Однак допоміжні та організаційні процеси (контроль вимог, управління якістю та ін) зазвичай виконуються паралельно з процесом розробки. У даній моделі повернення до початкового процесу передбачається після супроводу та виправлення помилок.

Особливість такої моделі полягає у фіксації послідовних процесів розробки програмного продукту. В її основу покладено модель фабрики, де продукт проходить стадії від задуму до виробництва, а потім передається замовнику як готовий продукт, зміну якого не передбачено, хоча можлива заміна на інший подібний продукт у випадку рекламації або деяких її деталей, що вийшли з ладу.

Недоліки цієї моделі:

• процес створення ПС не завжди вкладається в таку жорстку форму і послідовність дій;

• не враховуються потреби користувачів, що змінилися, зміни у зовнішньому середовищі, які викликають зміни вимог до системи в ході її розробки;

• великий розрив між часом занесення помилки (наприклад, на етапі проектування) і часом її виявлення (при супроводі), що призводить до великої переробки ПС.

При застосуванні каскадної моделі можуть мати місце такі фактори ризику:

• вимоги до ПС недостатньо чітко сформульовані, або не враховують перспективи розвитку ОС, середовищ і т.п.;

• велика система, яка не допускає компонентну декомпозицію, може спричинити проблеми з розміщенням її в пам'яті або на платформах, що не передбачені у вимогах;

• внесення швидких змін у технологію і у вимоги може погіршити процес розробки окремих частин системи або системи в цілому;

• обмеження на ресурси (людські, програмні, технічні та ін.) в ході розробки можуть звузити окремі можливості реалізації системи;

отриманий продукт може виявитися поганим для застосування з причини недостатнього розуміння розробниками вимог або функцій системи або недостатньо проведеного тестування. Переваги реалізації системи за допомогою каскадної моделі наступні:

• всі завдання підсистем та системи реалізуються одночасно (тобто жодна задача не забута), а це сприяє встановленню стабільних зв'язків і відносин між ними;

• повністю розроблену систему з документацією на неї легше супроводжувати, тестувати, фіксувати помилки і вносити зміни не безладно, а цілеспрямовано, починаючи з вимог (наприклад, додати або заміняти деякі функції) і повторити процес.

Каскадну модель можна розглядати як модель ЖЦ, придатну для створення першої версії ПЗ з метою перевірки реалізованих в ній функцій. При супроводі та експлуатації можуть бути виявлені різного роду помилки, виправлення яких потребують повторного виконання всіх процесів, починаючи з уточнення вимог.

Інкрементна модель ЖЦ

Перша створювана проміжна версія системи (випуск 1) реалізує частину вимог, в наступну версію (випуск 2) додають додаткові вимоги і так до тих пір, поки не будуть остаточно виконані всі вимоги та вирішені завдання розробки системи. Для кожної проміжної версії на етапах ЖЦ виконуються необхідні процеси, роботи та завдання, в тому числі, аналіз вимог та створення нової архітектури, які можуть бути виконані одночасно.

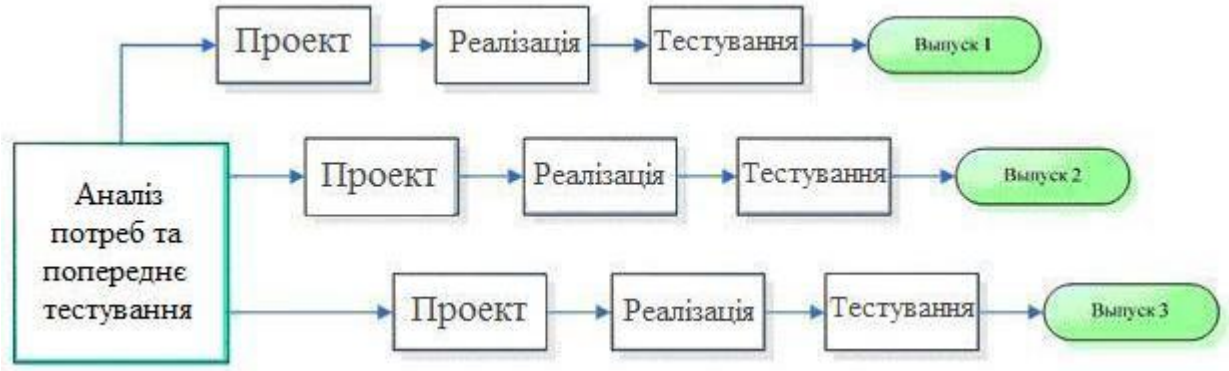

Рис**. 1.5.** Інкрементна модель ЖЦ

Процеси розробки технічного проекту ПС, його програмування та тестування, збирання та кваліфікаційні випробування ПС виконуються при створенні кожної наступної версії.

Згідно з цією моделлю ЖЦ, процеси якої практично такі ж, що і в каскадної моделі, орієнтир робиться на розробку деякої закінченої проміжної версії, а завдання процесу розробки виконуються послідовно або частково паралельно для ряду окремих проміжних структур версії.

Роботи та завдання процесу розробки наступної версії системи з додатковими вимогами або функціями можуть виконуватися неодноразово в тій же послідовності для всіх проміжних версій системи. Процеси супроводу та експлуатації можуть бути реалізовані паралельно з процесом розробки версії шляхом перевірки частково реалізованих вимог у кожній проміжній версії і так до отримання закінченого варіанту системи. Допоміжні та організаційні процеси ЖЦ зазвичай виконуються паралельно з процесом розробки версії системи і до кінця розробки будуть зібрані дані, на підставі яких може бути встановлений рівень завершеності та якості виготовленої системи.

При застосуванні даної моделі необхідно враховувати наступні фактори ризику:

• вимоги складені з урахуванням можливості їх зміни при реалізації продукту;

• всі можливості системи необхідно реалізувати з початку;

• швидке зміна технології та вимог до системи може призвести до порушення отриманої структури системи;

• обмеження в ресурсному забезпеченні (виконавці, фінанси) можуть призвести до затягування термінів здачі системи в експлуатацію.

Дану модель ЖЦ доцільно використовувати, у випадках коли:

• бажано реалізувати деякі можливості системи швидко за рахунок створення проміжної версії продукту;

• система декомпозується на окремі складові частини, які можна реалізовувати як деякі самостійні проміжні або готові продукти;

• можливе збільшення фінансування на розробку окремих частин системи.

Спіральні модель

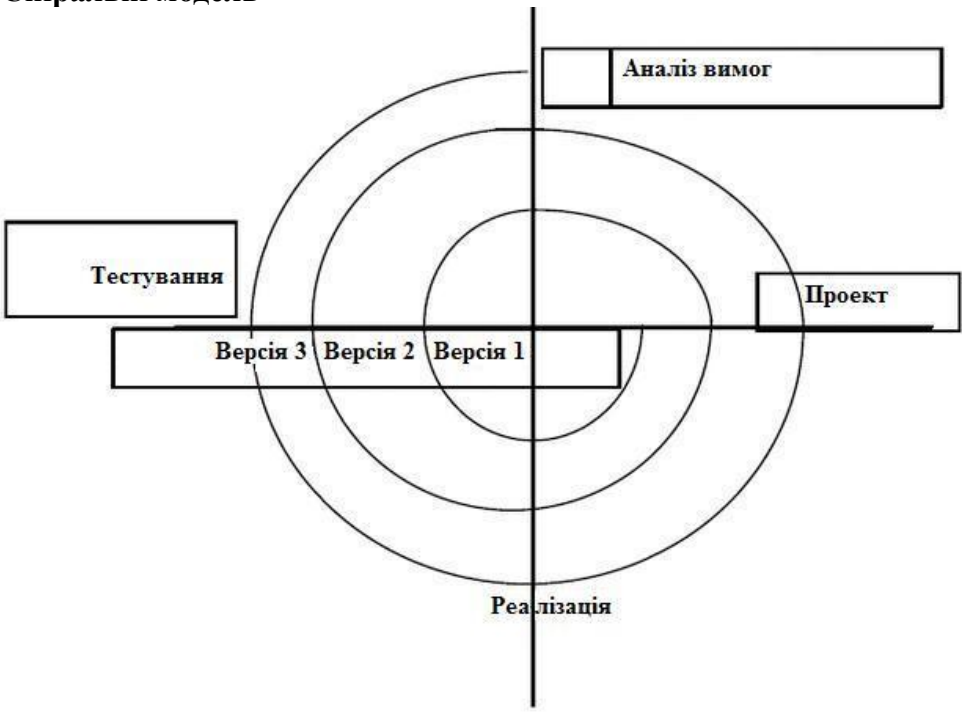

Рис. 2.6. Спіральні модель ЖЦ розробки програмних систем

Виходячи з можливості внесення змін, як до процесу, так і в створюваний проміжний продукт була створена спіральна модель (рис. 2.6). Внесення змін орієнтовано на задоволення потреби користувачів відразу, як тільки буде встановлено, що створені артефакти або елементи документації (опис вимог проекту, коментарі різного виду і т.п.), не відповідають дійсному стану розробки.

Дана модель ЖЦ допускає аналіз продукту на витку розробки, його перевірку, оцінку правильності та прийняття рішення рухатися на наступний виток або опуститися на попередній виток для доопрацювання на ньому проміжного продукту.

Відмінність цієї моделі від каскадної моделі полягає у можливості забезпечувати багаторазове повернення до процесу формулювання вимог і до повторної розробки з будь-якого процесу виконання робіт. На зображеній моделі (рис. 2.6), кожен виток спіралі відповідає одній з версій розробки системи.

При необхідності внесення змін в систему на кожному витку з метою отримання нової версії системи обов'язково вносяться зміни до попередньо зафіксованих вимог, після чого відбувається повернення на попередній виток спіралі для продовження реалізації нової версії системи з урахуванням змін.

Еволюційна модель ЖЦ

У випадку еволюційної моделі система розробляється у вигляді послідовності блоків структур (конструкцій). На відміну від інкрементної моделі ЖЦ мається на увазі, що вимоги встановлюються частково і уточнюються в кожному наступному проміжному блоці структури системи.

Використання еволюційної моделі припускає проведення дослідження предметної області для вивчення потреб замовника проекту та аналізу можливості застосування цієї моделі для реалізації. Модель застосовується для розробки нескладних і не критичних систем, для яких головною вимогою є реалізація функцій системи. При цьому вимоги не можуть бути визначені відразу і повністю. Тоді розробка системи проводиться ітераційний

шляхом її еволюційного розвитку з одержанням деякого варіанту системи - прототипу, на якому перевіряється реалізація вимог. Іншими словами, такий процес за своєю суттю є ітераційний, етапи розробки якого повторюються, починаючи від змінених вимог і до отримання готового продукту. У певному сенсі до цього типу моделі можна віднести спіральну модель.

Розвитком цієї моделі є модель еволюційного прототипування в рамках усього ЖЦ розробки (рис. 1.7). В літературі вона часто називається моделлю швидкої розробки додатків RAD (Rapid Application Development). У даній моделі наведені дії, які пов'язані з аналізом її застосування для конкретного виду системи, а також обстеження замовника для визначення потреб користувача для розробки плану створення прототипу.

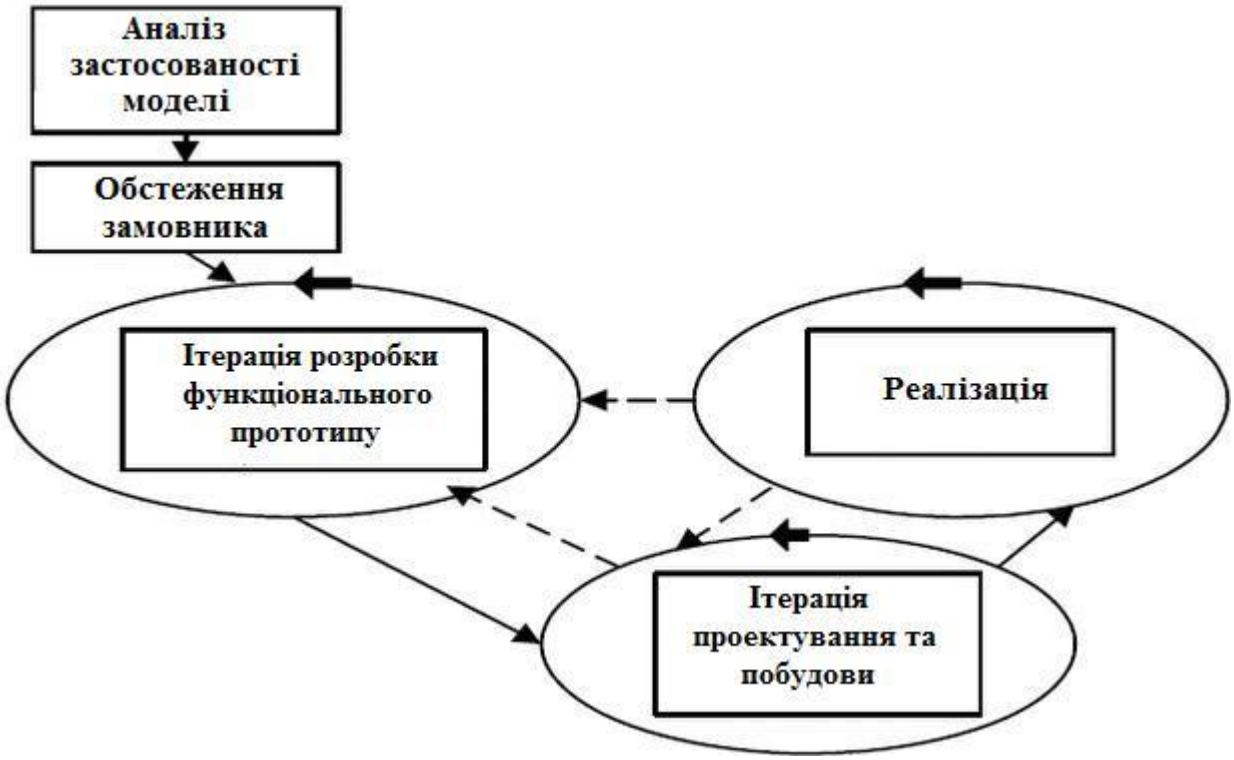

Рис. 1.7. Модель еволюційного прототипування

У моделі є дві головні ітерації розробки функціонального прототипу, проектування та реалізації системи. Перевіряється, задовольняє вона всім функціональним і не функціональні вимогам. Основною ідеєю цієї моделі є моделювання окремих функцій системи в прототипі і поступове еволюційна його доробка до виконання всіх заданих функціональних вимог.

Ітерацій по отриманню проміжних варіантів прототипу може бути декілька, в кожній з яких додається функція і повторно моделюються робота прототипу. І так до тих пір, поки не будуть промодельовані всі функції, задані у вимогах до системи. Потім виконується ще ітерація - остаточне програмування для отримання готової системи.

Ця модель застосовується для систем, в яких найбільш важливими є функціональні можливості, і які необхідно швидко продемонструвати на CASE-засоби.

Так як проміжні прототипи системи відповідають реалізації деяким функціональним вимогам, то їх можна перевіряти і при супроводі та експлуатації, тобто паралельно з процесом розробки чергових прототипів системи. При цьому допоміжні та організаційні процеси можуть виконуватися паралельно з процесом розробки та накопичувати відомості за даними кількісних і якісних оцінок в процесі розробки.

При цьому враховуються такі фактори ризику:

• реалізація всіх функцій системи одночасно може призвести до громіздкості;

• обмежені людські ресурси зайняті розробкою протягом тривалого часу.

Переваги застосування даної моделі ЖЦ наступні:

• швидка реалізація деяких функціональних можливостей системи та перевірка їх робото-придатності;

• використання проміжного продукту в наступному прототипі;

• виділення окремих функціональних частин для реалізації їх у вигляді прототипу;

• можливість збільшення фінансування системи;

• зворотній зв'язок встановлюється з замовником для уточнення функціональних вимог;

• спрощення внесення змін у зв'язку з заміною окремої функції.

Модель розвивається у напрямі долучення не функціональних вимог до системи, пов'язаних із захистом та безпекою даних, несанкціонованим доступом до них та ін.

Стандартизація моделі ЖЦ

Типовий ЖЦ системи починається з формулювання ідеї або потреби, проходить всі процеси розробки, виробництва, експлуатації та супроводу системи. Стандартний ЖЦ складається з процесів, кожен процес характеризується видами діяльності та завданнями, які виконуються на ньому. Перехід від одного процесу до іншого має бути санкціонований і визначені вхідні та вихідні дані.

Модель даного ЖЦ включає в себе процеси:

• визначення вимог;

• розробка (проектування, конструювання);

• верифікація, валідація, тестування;

• виготовлення;

• експлуатація;

• супровід.

Даній моделі відповідають всі види діяльності, починаючи з розробки проекту або концепції програмного продукту і закінчуючи його виготовленням. Як було сказано вище, стандарт ISO / IEC 12207 об'єднує ці види діяльності в наступні три категорії: основні, організаційні та допоміжні процеси, які і складають стандартний ЖЦ.

Процеси придбання, постачання та розробки використовуються для аналізу і визначення системних вимог та рішень верхнього рівня проектування системи та попереднього визначення вимог до компонентів системи, включаючи ПЗ. Процес розробки може бути використаний для аналізу, демонстрації, прототипування вимог і проектних рішень.

На етапі проектування розробляється технічне, програмне, організаційне забезпечення системи, а також проектуються, розробляються, інтегруються, тестуються і оцінюються її компоненти. Результатом цього процесу є система, яка розроблена відповідно до контракту або договору.

Стандарт розроблений так, щоб його можна було застосувати повністю або частково. Дії та завдання основних процесів відбираються, адаптуються і застосовуються при розробці або модифікації системи. Процес розробки може включати одну або більше ітерацій. Результатом є вимоги до ПЗ, проект і реалізований продукт.

Якщо розробляються ПЗ - частина системи, то до неї можуть застосовуватися всі дії процесів розробки, і якщо ця частина - автономне ПЗ, то деякі спільні дії на рівні системи можуть не використовуватися при його розробці.

Під час процесу виготовлення система готується для поставки замовнику і покупцям. Мета процесу - тиражування (виробництво) і установка працюючої системи у замовника для супроводження. Цей процес полягає в копіюванні виготовленого продукту та документації на відповідні носії користувачів. До видів діяльності на процесі відносяться досягнення якості реалізації та створення конфігурації (версії) системи. Інші допоміжні процеси і дії (наприклад, збір даних про результати контролю) можуть застосовуватися в міру необхідності.

Виготовлена система, починаючи з першої версії, передається замовнику або продається бажаючим покупцям. Інші процеси (придбання, постачання та розробки) можуть використовуватися при інсталяції і перевірці розробленої або модифікованої системи.

Процес експлуатації включає використання системи її покупцями. Коли система більше не задовольняє користувачів, вона утилізується, тобто видаляються з вживання шляхом знищення кодів, архівів, процедур і т.п.

Під час супроводу система модифікується внаслідок виявлених помилок і недоліків у її розробці або за вимогами користувача, який бажає її адаптувати до нового середовища або вдосконалити окремі її функції.

Зіставлення ЖЦ стандарту ISO / IEC 12207 та областей SWEBOK

Кожна область ядра знань SWEBOK по суті відповідає одному або декільком процесам, які визначені в стандарті ISO / IEC 12207. У зв'язку з цим проведено порівняльний аналіз областей SWEBOK і процесів моделі ЖЦ згаданого стандарту. Для цього спочатку розглянемо процеси ЖЦ, а потім області SWEBOK.

Характеристика процесів стандарту ISO / IEC 12207

Процеси даного стандарту розбиті за групами: основні, допоміжні та організаційні.

До основних процесів стандарту відносяться:

- придбання (acquisition);
- постачання (supply);
- розробка (development);
- експлуатація (operation);
- супровід (maintenance).

Процес придбання ініціює ЖЦ ПЗ і визначає дії організації-покупця (або замовника), яка набуває автоматизовану систему, програмний продукт чи сервіс.

Процес постачання визначає дії підприємства-постачальника, що забезпечує покупця системою, програмним продуктом чи сервісом.

Процес розробки полягає у виготовленні виконавцем проекту програмного продукту на процесах ЖЦ: розробка вимог, проектування, кодування, тестування та інтеграція.

Процес експлуатації визначає дії оператора з обслуговування системи, використання її користувачами, вивчили її можливості для задоволення своїх потреб в плані обробки даних або обчислень.

Процес супроводу полягає у виконанні запропонованих дій з інсталяції системи, запуску функцій, а також з управління модифікаціями і підтримкою системи в робочому стані.

До допоміжних процесів стандарту відносяться процеси:

- документування (documentation);
- управління конфігурацією (configuration management);
- забезпечення якості (quality assurance);
- верифікації (verification);
- валідації (validation);
- спільного аналізу (оцінки) (joint review);
- аудиту (audit).

Допоміжні процеси підтримують реалізацію основних процесів і сприяють отриманню необхідної якості ПЗ. Вони ініціюються іншими процесами.

До організаційних процесів стандарту відносяться процеси:

- управління (management);
- створення інфраструктури (infrastructure);
- удосконалення (improvement);
- навчання (training).

За кожним процесом стандарту спостерігає певний учасник розробки або керівник у частині виконання передбачених видів діяльності та завдання, які в нього входять, та перевірки результатів. У табл. 1.2. приведено загальна кількість визначених у стандарті процесів, дій і завдань.

Характеристика областей знань SWEBOK

У ядрі знань SWEBOK визначено 10 областей знань. Серед них виділимо базові галузі, методи та засоби яких відповідають процесам розробки ПС:

1. Розробка вимог;
2. Проектування;

3. Конструювання;
4. Тестування;
5. Супроводження.

Ці галузі знань по своїм базовим концепціям і методам, визначеним у SWEBOK, відповідають завданням і виконуваним діям наступних процесів розробки ЖЦ стандарту ISO / IEC - 12207:

1. Розробка вимог;
2. Проектування;
3. Кодування;
4. Тестування;
5. Інтеграція;
6. Інтеграційні тестування;
7. Експлуатація;
8. Супроводження.

Ці процеси задають послідовність завдань та дій при розробці різних типів ПС із застосуванням методів і засобів, які представлені в ядрі знань для перерахованих п'яти областей SWEBOK. Фактично процеси та області збігаються за змістом і назвою, але зміст дій на процесах визначаються методами і засобами п'яти областей, які наведені вище.

У табл. 1.2. наведено порівняльний перелік основних областей SWEBOK, їх завдань та відповідно до завдань ЖЦ стандарту.

Інші п'ять областей ядра SWEBOK відносяться до числа процесів забезпечення і управління розробкою проекту, при яких проводиться верифікація, збір даних для проведення оцінки якості та ін. І хоча області ядра знань явно не містять назв процесів ЖЦ, функціонально та змістовно вони відповідають процесам, що відносяться до категорії основних, допоміжних і організаційних.

Перелік процесів ЖЦ категорії допоміжних і організаційних наведено на рис.1.2, а відповідні їм області знань SWEBOK такі:

- управління конфігурацією,
- управління інженерією ПЗ (або управління проектом),
- процес інженерії ПЗ (інфраструктура процесу розробки),
- методи та засоби інженерії;
- інженерія якості (управління якістю).

Дані області знань включають методи та засоби розробки ПС, а також управління проектом, ризиками, конфігурацією, якістю створюваного продукту. Вони відповідають окремим завданням допоміжних і організаційних процесів ЖЦ стандарту і призначені для управління проектом, конфігурацією і якістю.

Таблиця 1.2. Завдання основних областей SWEBOK і процесів ЖЦ

Область SWEBOK	**Завдання області SWEBOK**	**Задачі процесів ЖЦ стандарту ISO / IEC 12207**
Розробка вимог	Інженерія вимог Виявлення вимог Аналіз вимог Специфікація вимог Перевірка вимог Управління вимогами	Підготовка замовлення Виявлення вимог Аналіз вимог до системи Аналіз вимог до ПЗ Опис документа
Проектування ПЗ	Розробка архітектури ПЗ Нотація Аналіз якості проектування Використання стратегії та методів проектування	Проектування: • архітектури системи • архітектури ПЗ • ПЗ Кодування ПЗ Тестування ПЗ
Конструювання	Зниження складності	Конструювання структури

ПЗ	Попередження відхилень від стилю Структуризація системи задля перевірок Використання зовнішніх стандартів	системи Кодування елементів структури и ПЗ Інтеграція елементів Застосування стандартів програмної інженерії
Тестування ПЗ	Тестування елементів і системі Тестування специфікацій, структури і системи на наборах даних Метричне вимірювання тестування Планування та оцінка якості	Тестування ПЗ Інтеграційне тестування Кваліфікаційне тестування Інтеграція системи Системне тестування Установка и приймання ПЗ
Супровід ПЗ	Запуск ПЗ Знаходження помилок, планування виправлень Внесення змін	Інсталяція ПЗ Аналіз проблем и модифікація Реалізація модифікацій Аналіз супроводу Міграція, видалення ПЗ
Експлуатація системи	Методи забезпечення експлуатації системи	Впровадження процесу Функціональне тестування Експлуатація системи Підтримка користувача

У табл. 1.3 наведено перелік областей ядра SWEBOK та відповідні завдання допоміжних (організаційних та додаткових) процесів ЖЦ стандарту ISO / IEC 12207.

Таблиця 1.3. Задачі областей SWEBOK и допоміжних процесів ЖЦ

Області SWEBOK	**Задачі областей SWEBOK**	**Задачі процесів стандарту 12207**
Управління конфігурацією	Процес управління конфігурацією. Ідентифікація елементів. Облік статусу, аудит. Контроль конфігурації. Управління версіями.	Визначення та контроль конфігурації. Облік стану та оцінка конфігурації. Управління реалізацією і поставкою версії.
Управління проектом	Організаційне управління. Планування проектом. Управління процесами и проектом. Інженерія вимірювань ПЗ. Управління ризиком.	Ініціація і визначення області застосування. Планування. Виконання та контроль. Аналіз управління проектом: • технічний аналіз; • аудит (ревізія).
Управління якістю	Концепція якості ПЗ. Визначення та планування якістю. Верифікація та валідація. Вимірювання в аналізі якості ПЗ.	Впровадження процесу. Забезпечення виробництва та якості. Процес верифікації та валідації. Аналіз и оцінка якості.
Методи та засоби інженерії	Методи інженерії. Інструменти інженерії.	Процес вдосконалювання: • визначення процесу; • оцінка процесу; - поліпшення процесу.
Процес інженерії ПЗ	Інфраструктура процесу. Визначення процесу.	Створення інфраструктури. Впровадження інфраструктури.

	Вимірювання процесу. Аналіз проекту. Виконання змін. Оцінки вартості і затрат.	Впровадження процесу. Завершення.

Зіставлення концепцій, методів і засобів областей SWEBOK з завданнями процесів ЖЦ дозволяє регламентувати пошук, виявлення помилок та внесення змін у вимоги до системи.

1.3.4. Екстремальне програмування

Реалії останніх років показали, що для систем, вимоги до яких змінюються достатньо часто, необхідно ще більше зменшити тривалість витка спірального життєвого циклу. У зв'язку з цим зараз стали вельми популярними швидкі життєві цикли розробки, наприклад, життєвий цикл в методології eXtreme Programming (XP).

Основна ідея життєвого циклу екстремального підходу - максимальне скорочення тривалості одного етапу життєвого циклу і тісна взаємодія із замовником. По суті, на кожному етапі відбувається реалізація і тестування однієї функції системи, після завершення яких система відразу передається замовникові на перевірку або експлуатацію.

Основна проблема даного підходу - інтерфейси між модулями, що реалізовують одну функцію. Якщо у всіх попередніх типах життєвого циклу інтерфейси достатньо чітко визначаються на самому початку розробки, оскільки заздалегідь відомі всі модулі, то при екстремальному підході інтерфейси проектуються "на льоту", разом з модулями, що розробляються.

1.3.5. Порівняння різних типів життєвого циклу і допоміжні процеси

Особливості розглянутих вище типів життєвого циклу зведені в таблицю 1.1. З неї можна бачити, що різні типи життєвих циклів застосовуються залежно від планованої частоти внесення змін в систему, термінів розробки і її складності. Життєві цикли з коротшими фазами більше підходять для розробки систем, вимоги до яких ще не визначені остаточно і виробляються у взаємодії із замовником системи під час її розробки.

Таблиця 1.1. Порівняння різних типів життєвого циклу

Тип життєвого циклу	**Довжина циклу**	**Верифікація і внесення змін**	**Інтеграція окремих компонент системи**
Каскадний	Всі етапи розробки системи. Довгий	В кінці розробки всієї системи. Зміни вносяться рідко	Чітко визначені до початку кодування інтерфейси
V-подібний	Всі етапи розробки системи. Довгий	В кінці повної розробки кожного з етапів системи. Зміни вносяться з середньою частотою	Рідко змінні інтерфейси
Спіральний	Розробка однієї версії системи. Середній	В кінці розробки кожного з етапів версії системи. Зміни вносяться з середньою частотою	Періодично змінні інтерфейси, що рідко міняються в межах версії
XP	Розробка однієї історії. Короткий	В кінці розробки кожної історії. Зміни вносяться дуже часто	Часто змінні інтерфейси

У приведеному вище описі різних моделей життєвого циклу по суті розглядався тільки один процес - процес розробки системи. Насправді в будь-якій моделі життєвого циклу можна побачити чотири види процесів:

1. Основний процес розробки
2. Процес верифікації
3. Процес управління розробкою
4. Допоміжні (підтримуючі) процеси

Процес верифікації - процес, направлений на перевірку коректності системи, що розробляється, і визначення ступеня її відповідності вимогам замовника.

Процес управління розробкою - окрема дисципліна. На управління дуже сильно впливає тип життєвого циклу основного процесу розробки. По суті, чим коротше один етап життєвого циклу, тим активніше управління і тим більше завдань стоїть перед менеджером проекту. При класичних схемах досить просто побудувати ієрархічну піраміду підлеглості, в якій кожен нижчестоячий менеджер відповідає за розробку певної частини системи. У XP-підході немає жорсткого розділення системи на частини, і менеджер повинен охоплювати всі етапи. При цьому процес управління активний впродовж всього життєвого циклу основного процесу розробки.

Допоміжні (підтримуючі) процеси забезпечують своєчасне створення всього, що може знадобитися розробникові або кінцевому користувачеві. Сюди входить підготовка призначеної для користувача документації, підготовка приймально-здавальних тестів, управління конфігураціями і змінами, взаємодія із замовником і т.д. Взагалі кажучи, допоміжні процеси можуть існувати протягом всього життєвого циклу розробки, а можуть бути своєрідними з'єднуючими ланками між процесом розробки і процесом експлуатації.

В даному курсі особлива увага буде приділена найбільш значущим підтримуючим процесам - процесу управління конфігураціями і процесу забезпечення гарантії якості.

Основна мета **процесу управління конфігураціями** - забезпечення цілісності всіх даних, що виникають в процесі колективної розробки. Під цілісністю розуміється, перш за все, ідентифікованість, доступність цих даних у будь-який момент часу і недопущення несанкціонованих змін. Важливим аспектом при цьому стає процес управління змінами даних, тобто планування і затвердження будь-яких змін в проектну документацію або програмний код, а також визначення області впливу цих змін.

Процес гарантії якості забезпечує проведення перевірок, що гарантують, що процес розробки задовольняє набору певних вимог (стандартів), необхідних для випуску якісної продукції. Фактично він перевіряє, що всі передбачені стандартами розробки процедури виконуються і при виконанні дотримуються ті, що декларують для них правила.

Потрібно особливо відзначити, що процес гарантії якості не гарантує розробку якісної програмної системи. Він гарантує тільки, що процеси розробки побудовані і виконуються так, щоб не знижувати якість продукції.

Вимоги, які пред'являються до організації роботи, необхідної для випуску якісної продукції, оформлені у вигляді стандартів якості. Найбільш часто використовувана група стандартів якості - ISO 9126. На додаток до них існує стандарт, що містить вимоги до життєвого циклу розробки ПЗ, - ISO 12207.

У реальній практиці зараз найширше застосовується стандарт ISO 12207, у вітчизняних держструктурах використовуються стандарти серії ГОСТ 34.

Стандарти комплексу ГОСТ 34 на створення і розвиток ПС - узагальнені, ісприймаються як вельми жорсткі по структурі ЖЦ і проектній документації.

Міжнародний стандарт ISO/IEC 12207 організації життєвого циклу продуктів програмного забезпечення (ПЗ) містить загальні рекомендації по організації життєвого циклу.

1.2. Сучасні технології розробки програмного забезпечення

1.2.1. Microsoft Solutions Framework

Microsoft Solutions Framework (MSF) - це методологія ведення проектів і розробки рішень, що базується на принципах роботи над продуктами фірми Microsoft і призначена для використання в організаціях, що потребують концептуальної схеми для побудови сучасних рішень.

Microsoft Solutions Framework є схемою для ухвалення рішень по плануванню і реалізації нових технологій в організаціях. MSF включає навчання, інформування, рекомендації і інструменти для ідентифікації і структуризації інформаційних потоків бізнес-процесів і всієї інформаційної інфраструктури нових технологій.

Microsoft Solutions Framework є добре збалансованим набором методик організації процесу розробки, який може бути адаптований під потреби практично будь-якого колективу розробників. MSF містить не тільки рекомендації загального характеру, але і пропонує модель колективу розробників, що адаптується, визначає взаємини усередині колективу, гнучку модель проектного планування, заснованого на управлінні проектними групами, а також набір методик для оцінки ризиків.

MSF складається з двох моделей:

- модель проектної групи;
- модель процесів

і трьох дисциплін:

- управління проектами;
- управління ризиками;
- управління підготовкою.

У MSF немає ролі "менеджер проекту" і ієрархії керівництва, управління розробкою розподілене між керівниками окремих проектних груп усередині колективу, що виконують наступні завдання:

- Управління програмою
- Розробка
- Тестування
- Управління випуском
- Задоволення споживача
- Управління продуктом

Життєвий цикл процесів в MSF поєднує водоспадну і спіральну моделі розробки: проект реалізується поетапно, з наявністю відповідних контрольних точок, а сама послідовність етапів може повторюватися по спіралі (Рис. 1.8).

Рис. 1.8. Життєвий цикл в MSF

При такому підході від моделі водоспаду береться простота планування, від класичної спіральної - легкість модифікацій. Завдяки проміжним контрольним точкам і зворотній спіралі верифікація полегшується взаємодія із замовником.

При управлінні проектом чітко ставиться мета, яку необхідно досягти в результаті, і враховуються обмеження, що накладаються на проект. Всі види обмежень можуть бути віднесені до одного з трьох видів:

обмеження ресурсів,

обмеження часу

і обмеження можливостей.

Ці три види обмежень і пріоритетність завдань по їх подоланню утворюють трикутник пріоритетів в MSF (Рис. 1.9).

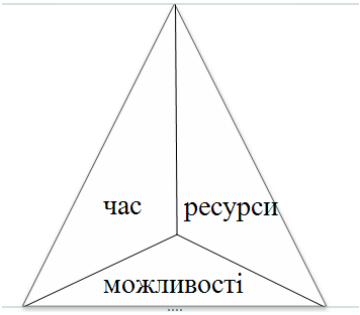

Рис. 1.9. Трикутник пріоритетів в MSF

Трикутник пріоритетів є основою для матриці компромісів - заздалегідь затверджених уявлень про те, які аспекти процесу розробки будуть чітко задані, а які узгоджуватимуться.

Microsoft випустила середовище розробки, що повною мірою підтримує основні ідеї MSF, - Visual Studio Team System 2010. Це програмний комплекс, що є не середовищем розробки для індивідуальних членів колективу, а комплексним засобом підтримки колективної роботи.

- Об'єднання розстановки пріоритетів і управління IT-проектами за рахунок інтеграції з Microsoft Office Project Server
- Управління проектом як множиною проектів для забезпечення проактивного балансування розподілу ресурсів відповідно до бізнес-пріоритетів.
- Можливість повного відстежування (включаючи ієрархію робочих елементів) для стеження за ключовими параметрами проекту на предмет відповідності вимогам, а також можливість здійснювати швидкий аналіз змін, що відбуваються.
- Всесторонні вимірювання і «датчики» для загальної візуалізації статусу проекту і його прогресу по ключових параметрах
- Нові могутні інструменти, що дозволяють розробникам і тестувальникам проводити швидку ідентифікацію, обговорювати, розставляти пріоритети, діагностувати і виправляти помилки.
- Вбудоване управління наборами тестів для створення, впорядковування і управління наборами тестів для команд розробників і тестувальників.
- Автоматизація тестування і його управління для полегшення концентрації розробників і тестувальників перш за все на бізнес-рівні тестування, а не на тестуванні, що повторюється або ручному
- Вимірювання якості для ухвалення рішення про випуск або відкладання продукту залежно від того, чи готове застосування для використання споживачем і чи пройшло воно повне тестування на предмет відповідності запитам бізнесу.
- Швидке включення видалених, розподілених, таких, що не мають доступу або аутсорсних команд в процес розробки
- Легке налаштування процесу і інструкція від Microsoft і партнерів при пошуку схожих методів роботи з необхідними вам
- Поліпшення в мультисерверной адміністрації, збірки і контролю коду
- Test Impact Analysis допоможе розробникам швидко вносити зміни (check-in) коди при перевірці проходження заздалегідь визначеного набору ключових тестів.

1.2.2. Rational Unified Process

Rational Unified Process - це методологія створення програмного забезпечення, оформлена у вигляді розміщуваної на Web бази знань, яка забезпечена пошуковою системою.

Продукт Rational Unified Process (RUP) розроблений і підтримується Rational Software. Він регулярно оновлюється з метою обліку передового досвіду і поліпшується за рахунок перевірених на практиці результатів.

RUP забезпечує строгий підхід до розподілу завдань і відповідальності усередині організації-розробника. Його призначення полягає в тому, щоб гарантувати створення вчасно

і в рамках встановленого бюджету якісного ПЗ, що відповідає потребам кінцевих користувачів.

RUP сприяє підвищенню продуктивності колективної розробки і надає краще з накопиченого досвіду по створенню ПЗ, за допомогою керівництва, шаблонів і повчань по користуванню інструментальними засобами для всіх критично важливих робіт, протягом життєвого циклу створення і супроводу ПЗ. Забезпечуючи кожному членові групи доступ до тієї ж самої бази знань, незалежно від того, чи розробляє він вимоги, проектує, виконує тестування або управляє проектом - RUP гарантує, що всі члени групи використовують спільну мову моделювання і процес, мають узгоджене бачення того, як створювати ПЗ. Як мова моделювання в загальній базі знань використовується Unified Modeling Language (UML), що є міжнародним стандартом.

Особливістю RUP є те, що в результаті роботи над проектом створюються і удосконалюються моделі. Замість створення величезної кількості паперових документів, RUP спирається на розробку і розвиток семантично збагачених моделей, що всесторонньо представляють систему, що розробляється. RUP - це керівництво по тому, як ефективно використовувати UML. Стандартна мова моделювання, використовувана всіма членами групи, робить зрозумілими для всіх описи вимог, проектування і архітектуру системи.

RUP підтримується інструментальними засобами, які автоматизують багато елементів процесу розробки. Вони використовуються для створення і вдосконалення різних проміжних продуктів на різних етапах процесу створення ПЗ, наприклад, при візуальному моделюванні, програмуванні, тестуванні і так далі

RUP - це процес, що конфігурується, оскільки цілком зрозуміло, що неможливо створити єдиного керівництва на всі випадки розробки ПЗ. RUP придатний як для маленьких груп розробників, так і для великих організацій, що займаються створенням ПЗ. У основі RUP лежить проста і зрозуміла архітектура процесу, яка забезпечує спільність для цілого сімейства процесів. Більш того, RUP може конфігуруватися для обліку різних ситуацій. У його склад входить Development Kit, який забезпечує підтримку процесу конфігурації під потреби конкретних організацій.

RUP описує, як ефективно застосовувати комерційно обгрунтовані і практично випробувані підходи до розробки ПЗ для колективів розробників, де кожен з членів отримує переваги від використання передового досвіду в:

- ітераційній розробці ПЗ;
- управлінні вимогами;
- використанні компонентної архітектури;
- візуальному моделюванні;
- тестуванні якості ПЗ;
- контролі за змінами в ПЗ.

RUP організовує роботу над проектом в термінах послідовності дій (workflows), продуктів діяльності, виконавців і інших статичних аспектів процесу, з одного боку, і в термінах циклів, фаз, ітерацій і тимчасових відміток завершення певних етапів в створенні ПЗ (milestones), тобто в термінах динамічних аспектів процесу - з іншою.

1.2.3. eXtreme Programming

Екстремальне програмування - порівняно молода методологія розробки програмних систем, заснована на поступовому поліпшенні системи і розробки її дуже короткими ітераціями. За своєю суттю екстремальне програмування (XP) - це одна з так званих "гнучких" методологій розробки ПЗ, яка є невеликим набором конкретних правил, що дозволяють максимально ефективно виконувати вимоги сучасної теорії управління програмними проектами.

XP орієнтована на:

- командну роботу з тісними зв'язками усередині команди і із замовником;
- розробку найбільш простих працюючих рішень;
- гнучке адаптивне планування;
- оперативний зворотний зв'язок (шляхом модульного і функціонального тестування).

Основними принципами XP є розробка невеликими ітераціями на підставі порції вимог замовника (т.з. призначених для користувача історій), написання функціональних тестів до написання програмного коду, постійне спілкування і постійний рефакторінг коду.

Основними практиками XP є:

- Планування процесу
- Часті релізи
- Проста архітектура
- Тестування
- Рефакторінг
- Парне програмування
- Колективне володіння кодом
- Часта інтеграція
- 40-годинний робочий тиждень
- Стандарти кодування
- Тісна взаємодія із замовником

1.2.4. Порівняння технологій MSF, RUP і XP

Основні особливості MSF, RUP і XP зведені в таблицю 1.5. По ній можна судити, що Rational Unified Process є добре збалансованим рішенням для середніх по розмірах колективів розробників, що працюють із застосуванням продуктів і технологій компанії Rational. Супровід розробки системи і самої системи регламентується методологією RUP, проте дана технологія достатньо сильно орієнтована на внутрішньофірмові інструментальні засоби.

Extreme Programming добре підходить для проектних груп малого розміру і для невеликих систем з часто змінними вимогами. Основна проблема XP - супроводжуваність. У разі зміни кадрів в колективі розробників значна частина проектної інформації може бути загублена через практично відсутню документацію.

Таблиця 1.5. Технології MSF, RUP і XP

Технологія	Оптимальна команда	Відповідність стандартам	Допустимі технології і інструменти	Зручність модифікації і супроводу
Rational Unified Process	10 - 40 чол.	стандарти Rational	UML і продукти Rational	Зручно (RUP)
Microsoft Solutions Framework	3 - 20 чол.	адаптована	будь-які	Зручно (MSF+MOF)
XP	2 - 10 чол.	стандарти відсутні	будь-які	Складно (залежність від конкретних учасників колективу)

Microsoft Solutions Framework є найбільш збалансованою технологією, орієнтованою на проектні групи малих і середніх розмірів. MSF не накладає ніяких обмежень на використовуваний інструментарій і містить рекомендації вельми загального характеру. Проте, ці рекомендації можуть бути використані для побудови конкретного процесу, відповідного потребам колективу розробників.

1.3. Ролевий склад колективу розробників, взаємодія між ролями в різних технологічних процесах

Коли проектна команда включає більше двох чоловік неминуче встає питання про розподіл ролей, має рацію і відповідальності в команді. Конкретний набір ролей визначається багатьма чинниками - кількістю учасників розробки і їх особистими

перевагами, прийнятою методологією розробки, особливостями проекту і іншими чинниками. Практично в будь-якому колективі розробників можна виділити перераховані нижче ролі. Деякі з них можуть бути зовсім відсутніми, при цьому окремі люди можуть виконувати відразу декілька ролей, проте загальний склад міняється мало.

Замовник (заявник). Ця роль належить представникові організації, що замовила систему, що розроблялася. Зазвичай заявник обмежений в своїй взаємодії і спілкується тільки з менеджерами проекту і фахівцем з сертифікації або впровадження. Зазвичай замовник має право змінювати вимоги до продукту (тільки у взаємодії з менеджерами), читати проектну і сертифікаційну документацію, що зачіпає нетехнічні особливості системи, що розробляється.

Менеджер проекту. Ця роль забезпечує комунікаційний канал між замовником і проектною групою. Менеджер продукту управляє очікуваннями замовника, розробляє і підтримує бізнес-контекст проекту. Його робота не пов'язана безпосередньо з продажем, він сфокусирован на продукті, його завдання - визначити і забезпечити вимоги замовника. Менеджер проекту має право змінювати вимоги до продукту і фінальну документацію на продукт.

Менеджер програми. Ця роль управляє комунікаціями і взаєминами в проектній групі, є в деякому роді координатором, розробляє функціональні специфікації і управляє ними, веде графік проекту і звітує за станом проекту, ініціює ухвалення критичних для ходу проекту рішень.

Менеджер програми має право змінювати функціональні специфікації верхнього рівня, план-графік проекту, розподіл ресурсів по завданнях. Часто на практиці роль менеджера проекту і менеджера програми виконує одна людина.

Розробник. Розробник ухвалює технічні рішення, які можуть бути реалізовані і використані, створює продукт, що задовольняє специфікаціям і очікуванням замовника, консультує інші ролі в ході проекту. Він бере участь в оглядах, реалізує можливості продукту, бере участь в створенні функціональних специфікацій, відстежує і виправляє помилки за прийнятний час. У контексті конкретного проекту роль розробника може мати на увазі, наприклад, інсталяцію програмного забезпечення, настройку продукту або послуги. Розробник має доступ до всієї проектної документації, включаючи документацію по тестуванню, має право на зміну програмної коди системи в рамках своїх службових обов'язків.

Фахівець з тестування. Фахівець з тестування визначає стратегію тестування, тест-требования і тест-планы для кожної з фаз проекту, виконує тестування системи, збирає і аналізує звіти про проходження тестування. Тест-требованія повинні покривати системні вимоги, функціональні специфікації, вимоги до надійності і здатності навантаження, призначені для користувача інтерфейси і власне програмний код. У реальності роль фахівця з тестування часто розбивається на дві - розробника тестів і тестувальника. Тестувальник виконує всі роботи по виконанню тестів і збору інформації, розробник тестів - всю решта робіт.

Фахівець з контролю якості. Ця роль належить членові проектної групи, який здійснює взаємодію з розробником, менеджером програми і фахівцями з безпеки і сертифікації з метою відстежування цілісної картини якості продукту, його відповідності стандартам і специфікаціям, передбаченим проектною документацією. Слід розрізняти фахівця з тестування і фахівця з контролю якості. Останній не є членом технічного персоналу проекту, відповідальним за деталі і техніку роботи. Контроль якості має на увазі насамперед контроль самих процесів розробки і перевірку їх відповідності визначеним в стандартах якості критеріям.

Фахівець з сертифікації. При розробці систем, до надійності яких пред'являються підвищені вимоги, перед введенням системи в експлуатацію потрібне підтвердження з боку уповноваженого органу (зазвичай державного) відповідності її експлуатаційних характеристик заданим критеріям. Така відповідність визначається в ході сертифікації системи. Фахівець з сертифікації може або бути представником сертифікуючих органів, включеним до складу колективу розробників, або навпаки - представляти інтереси

розробників в сертифікуючому органі. Фахівець з сертифікації приводить документацію на програмну систему у відповідність вимогам сертифікуючого органу або бере участь в процесі створення документації з урахуванням цих вимог. Також фахівець з сертифікації відповідальний за всю взаємодію між колективом розробників і сертифікуючим органом. Важливою особливістю ролі є незалежність фахівця від проектної групи на всіх етапах створення продукту. Взаємодія фахівця з членами проектної групи обмежується менеджерами за проектом і за програмою.

Фахівець з впровадження і супроводу. Бере участь в аналізі особливостей майданчика замовника, на якому планується проводити впровадження системи, що розробляється, виконує весь спектр робіт по установці і настройці системи, проводить навчання користувачів.

Фахівець з безпеки. Даний фахівець відповідальний за весь спектр питань безпеки створюваного продукту. Його робота починається з участі в написанні вимог до продукту і закінчується фінальною стадією сертифікації продукту.

Інструктор. Ця роль відповідає за зниження витрат на подальший супровід продукту, забезпечення максимальної ефективності роботи користувача. Важливо, що мова йде про продуктивності користувача, а не системи. Для забезпечення оптимальної продуктивності інструктор збирає статистику по продуктивності користувачів і створює вирішення для підвищення продуктивності, зокрема з використанням різних аудіовізуальних засобів. Інструктор бере участь у всіх обговореннях призначеного для користувача інтерфейсу і архітектури продукту.

Технічний письменник. Особа, що здійснює цю роль, несе обов'язки по підготовці документації до розробленого продукту, фінального опису функціональних можливостей. Також він бере участь в написанні супровідних документів (системи допомоги, керівництво користувача).

1.4. Приклад. ЖЦ розробки ПЗ із завданнями і діями задля процесу тестування.

Основне призначення процесу тестування ЖЦ - виконання завдань процесу на основі входів (вхідні дані для виконання завдань процесу) та виходів при завершенні завдань, а також ролей та дій виконавців цих завдань.

У відповідності зі стандартом ISO / IEC 12207 були виявлені завдання тестування і розподілені по процесам ЖЦ ПЗ. В результаті було отримано єдиний безперервний процес тестування різних ПЗ, завданнями якого є підготовка, проведення та оцінювання результатів тестування, які розподілилися по 20 діям (крокам) процесу розробки. Даний підхід до ретельному тестування ПЗ доцільно застосовувати, наприклад, для систем реального часу.

На кроці підготовки здійснюється аналіз робочих продуктів процесу розробки ПЗ (вхідних для даного кроку процесу тестування) для визначення цілей, об'єктів, сценаріїв і ресурсів тестування, адекватних кроку тестування. Результати виконання кроків підготовки тестування повинні фіксуватися в планах тестування.

Рис. 1.9. ЖЦ розробки ПЗ з конкретизованими завданнями на підпроцесах тестування

На кроці *виконання* здійснюється фіксація результатів виконання тестів, їх порівняння з очікуваними результатами, визначення поточного стану робочого продукту ПС і прийняття рішення про достатність тестування.

Кожен крок процесу *розроблення* складається з набору розв'язуваних задач, їх розподілу за процесами і підпроцесами ЖЦ. Кроки процесу й окремих задач можуть виконуватися циклічно для різних об'єктів ПС при їх тестуванні.

Опис семантики завдань і кроків процесу тестування представлено в табл. 1.1.

Таблиця 1.1. Склад задач процесу тестування

Крок процесу	**Завдання процесу тестування**
1. Створення групи тестування	1.1. Визначення учасників процесу тестування
	1.2. Розподіл обов'язків у групі і формування плану тестування
2. Аналіз ризику	2.1. Ідентифікація ризиків
	2.2. Впорядкування ризиків
	2.3. Розподіл ресурсів
3. Визначення цілей тестування	3.1. Ідентифікація тестування
	3.2. Визначення критеріїв проходження тестів
	3.3. Приведення в порядок цілей тестування за оцінками ризику
4. Розробка планів тестування	4.1. Розробка плану тестування ПС
	4.2. Розробка плану інтеграційного тестування
	4.3. Розробка плану автономного тестування
	4.4. Розробка плану комплексного тестування
5. Розробка тестів	5.1. Проектування та розробка тестів
	5.2. Підготовка тестових даних
	5.3. Перевірка тестових документів

6. Автономне та інтеграційні тестування	6.1. Автономне тестування модулів та аналіз результатів
	6.2. Інтеграційні тестування
	6.3. Повторне тестування після усунення дефектів
	6.4. Аналіз результатів інтеграційного тестування
7. Тестування ПС	7.1. Затвердження середовища і ресурсів тестування
	7.2. Тестування ПС
	7.3. Повторне тестування ПС після усунення дефектів
	7.4. Аналіз результатів завершення тестування ПС
	7.5. Тестування інсталяції ПС
8. Складання документа з тестування ПС та підготовка звіту	8.1. Збір та аналіз даних про результати тестування
	8.2. Підготовка рішень і рекомендацій з використання ПС
	8.3. Підготовка підсумкового документа про результати тестування
	8.4. Перевірка рішень і підготовка документа звіту

Для підключення задач тестування до всіх процесів ЖЦ проводиться:

– розподіл обов'язків між учасниками процесу з урахуванням вимог щодо їх професійної підготовки;

– визначення стандартів на подання остаточних документів, метрик процесу, критеріїв початку і завершення задач і переходу до наступного кроку процесу;

– підбір методів тестування для вибраного класу ПС і перевірки правильності виконання задач тестування;

– розроблення спеціальних шаблонів для документування процесу тестування щодо кожного його кроку.

При завершенні тестування ПС для визначення часу тестування, вартості робіт враховуються результати тестування процесу розробки ПС і оформлюється звітний документ по виготовленню ПС. Оцінювання ризику відмов проводиться на етапі підготовки тестування і на кроках аналізу.

Контрольні питання і завдання

1. Наведіть базові поняття області знань «Тестування ПЗ».
2. Визначте мету і задачі області знань «Інженерія якості ПЗ».
3. Вкажіть, який зв'язок існує між ядром знань SWEBOK і стандартом ЖЦ.

ТЕМА 2. ЗАБЕЗПЕЧЕННЯ ЯКОСТІ — ОСНОВНІ ПОНЯТТЯ І ВИЗНАЧЕННЯ

2.1. Завдання і цілі процесу верифікації

Спочатку розглянемо цілі верифікації. Основна мета процесу - доказ того, що результат розробки відповідає пред'явленим до нього вимогам. Зазвичай процес верифікації проводиться зверху вниз, починаючи від загальних вимог, заданих в технічному завданні і/або специфікації на всю інформаційну систему, і закінчуючи детальними вимогами до програмних модулів і їх взаємодії. До складу завдань процесу входить послідовна перевірка того, що в програмній системі:

- загальні вимоги до інформаційної системи, призначені для програмної реалізації, коректно перероблені в специфікацію вимог високого рівня до комплексу програм, що задовольняють початковим системним вимогам;
- вимоги високого рівня правильно перероблені в архітектуру ПЗ і в специфікації вимог до функціональних компонентів низького рівня, які задовольняють вимогам високого рівня;
- специфікації вимог до функціональних компонентів ПЗ, розташованим між компонентами високого і низького рівня, задовольняють вимогам більш високого рівня;
- архітектура ПЗ і вимоги до компонентів низького рівня коректно перероблені в тих, що задовольняють ним початкові тексти програмних і інформаційних модулів;
- початкові тексти програм і відповідний їм виконуваний код не містять помилок.

Крім того, верифікації на відповідність специфікації вимог на конкретний проект програмного засобу підлягають вимоги до технологічного забезпечення життєвого циклу ПЗ, а також вимоги до експлуатаційної і технологічної документації.

Цілі верифікації ПЗ досягаються за допомогою послідовного виконання комбінації з інспекцій проектної документації і аналізу їх результатів, розробки тестових планів тестування і тест-вимог, тестових сценаріїв і процедур і подальшого виконання цих процедур. Тестові сценарії призначені для перевірки внутрішньої несуперечності і повноти реалізації вимог. Виконання тестових процедур повинне забезпечувати демонстрацію відповідності випробовуваних програм початковим вимогам.

На вибір ефективних методів верифікації і послідовність їх застосування найбільшою мірою впливають основні характеристики тестованих об'єктів:

- клас комплексу програм, що визначається глибиною зв'язку його функціонування з реальним часом і випадковими діями із зовнішнього середовища, а також вимоги до якості обробки інформації і надійності функціонування;
- складність або масштаб (об'єм, розміри) комплексу програм і його функціональних компонентів, що є кінцевими результатами розробки;
- переважаючі елементи в програмах: здійснюючі обчислення складних виразів і перетворення вимірюваних величин або оброблювальні логічні і символьні дані для підготовки і відображення рішень.

Визначимо деякі поняття і визначення, пов'язані з процесом тестування як складовій частині верифікації. Майерс дає наступні визначення основних термінів [2].

Тестування - процес виконання програми з метою виявлення помилки.

Тестові дані - входи, які використовуються для перевірки системи.

Тестова ситуація (test case) - входи для перевірки системи і передбачувані виходи залежно від входів, якщо система працює відповідно до специфікації вимог.

Хороша тестова ситуація - та ситуація, яка володіє великою вірогідністю виявлення поки що невиявленої помилки.

Вдалий тест - тест, який виявляє поки що невиявлену помилку.

Помилка - дія програміста на етапі розробки, що приводить до того, що в програмному забезпеченні міститься внутрішній дефект, який в процесі роботи програми може привести до неправильного результату.

Відмова - непередбачувана поведінка системи, що приводить до неочікуваного результату, яке могло бути викликане дефектами, що міститься в ній.

Таким чином, в процесі тестування програмного забезпечення, як правило, перевіряють наступне:

- програмне забезпечення відповідає вимогам;
- у ситуаціях, не відбитих у вимогах, програмне забезпечення поводиться адекватно, тобто не відбувається відмова системи;
- наявність типових помилок, які роблять програмісти.

Забезпечення якості (Quality Assurance — QA) — це сукупність заходів, що охоплюють всі технологічні етапи розробки, випуску і експлуатації програмного забезпечення (ПЗ) інформаційних систем, що відбуваються на різних стадіях життєвого циклу ПЗ, для забезпечення якості продукту, що випускається.

Контроль якості (Quality Control — QC) — це сукупність дій що проводяться над об'єктом тестування в процесі розробки для отримання інформації про реальний стан об'єкту тестування в розрізах: "готовність Продукту до випуску", "Відповідність зафіксованим вимогам", "Відповідність заявленому рівню якості продукту".

Тестування програмного забезпечення (Software Testing) — це одна з технік контролю якості, що включає дії по плануванню робіт (Test Management), проектуванню тестів (Test Design), виконанню тестування (Test Execution) і аналізу отриманих результатів (Test Analysis).

Верифікація (verification) — це процес оцінки системи або її компонентів з метою визначення чи задовольняють результати поточного етапу розробки умовам, сформованим на початку цього етапу [IEEE]. Тобто чи виконуються наші цілі, терміни, завдання по розробці проекту, визначені на початку поточної фази.

Валідация (validation) — це визначення відповідності ПЗ, що розробляється очікуванням і потребам користувачів, вимогам до системи [BS7925-1].

Щодня в своїй роботі ми стикаємося з достатньо абстрактним поняттям «Якість ПЗ» і якщо поставити питання тестувальникові або програмістові — «що таке якість?», то у кожного знайдеться своє тлумачення. Розглянемо визначення "якості ПЗ" в контексті міжнародних стандартів:

[1061-1998 IEEE Standard for Software Quality Metrics Methodology]

Якість програмного забезпечення - це ступінь, в якому ПЗ володіє необхідною комбінацією властивостей.

[ISO 8402:1994 Quality management and quality assurance]

Якість програмного забезпечення — це сукупність характеристик ПЗ, що відносяться до його здатності задовольняти встановлені і передбачувані потреби.

На даний момент найбільш поширена і використовується багаторівнева модель якості програмного забезпечення, представлена в наборі стандартів **ISO 9126.** На верхньому рівні виділено **6** основних характеристик якості ПЗ, кожну з яких визначають набором атрибутів, що мають відповідні метрики для подальшої оцінки.

2.2. Модель якості програмного забезпечення

На даний момент найбільшпоширена багаторівнема модель якості ПЗ, представлена в наборі стандартів **ISO 9126.** На верхньому рівні виділено **6 основних характеристик якості ПЗ**, кожну з яких визначають набором атрибутів, які мають відповідні метрики для їх оцінки.

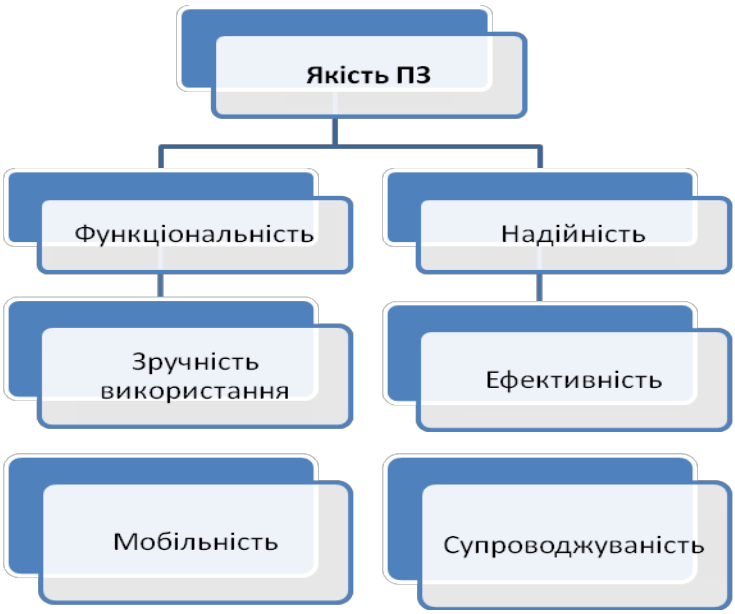

Рис.2.1. Модель якості ПЗ (ISO 9126-1)

Якість ПЗ

Функціональність

- -придатність до певної роботи
- -точність
- - здатність до взаємодії
- -відповідність стандартам
- - захищеність

Зручність використання

- - Зрозумілість
- - Зручність вивчення
- - Працездатність
- - Привабливість
- - Відповідність стандартам практичності

Надійність

- -зрілість
- -стійкість до відмов
- - здатність до відновлення
- - відповідність стандартам надійності

Ефективність

- **-**Часові характеристики
- -використання ресурсів
- - Відповідність стандартам ефективності

Супроводжуваність

- -аналізованість
- -зручність внесення змін
- -стабільність
- -зручність перевірки
- - відп. стандартам

Мобільність

- - адаптованість
- - зручність встановлення
- -здатність до співіснування з іншими ПЗ
- - зручність заміни
- - відповідність стандартам

• **Функціональність** (Functionality) — визначається здатністю ПЗ вирішувати завдання, які відповідають зафіксованим і передбачуваним потребам користувача, за заданих умов використання ПЗ. Тобто ця характеристика відповідає за те, що ПЗ працює справно і точно, функціонально сумісно, відповідає стандартам галузі і захищено від несанкціонованого доступу.

• **Надійність** (Reliability) — здатність ПЗ виконувати необхідні завдання в позначених умовах впродовж заданого проміжку часу або вказану кількість операцій. Атрибути даної характеристики — це завершеність і цілісність всієї системи, здатність самостійно і коректно відновлюватися після збоїв в роботі, відмовостійка.

• **Зручність використання** (Usability) — можливість легкого розуміння, вивчення, використання і привабливості ПЗ для користувача.

• **Ефективність** (Efficiency) — здатність ПЗ забезпечувати необхідний рівень продуктивності у відповідність з виділеними ресурсами, часом і іншими позначеними умовами.

• **Зручність супроводу** (Maintainability) — легкість, з якою ПЗ можна аналізувати, тестувати, змінювати для виправлення дефектів та реалізації нових вимог, для полегшення подальшого обслуговування і адаптувати до середовища.

• **Портативність**, мобільність (Portability) — характеризує ПЗ з погляду легкості його перенесення з одного оточення (software/hardware) в інше.

2.3.Забезпечення якості

Забезпечення якості (Quality Assurance) – це головним чином процес вивчення: вивчення того, що працює погано, і що ви можете зробити щодо цього; вивчення того, як працює середовище, в якому воно працює, і застосування цього як сталої практики; і вивчення того, як працювати краще з кожним проектом, який ви берете на свою відповідальність.

Організація забезпечення якості починається з наступних кроків. Їх відносно легко реалізувати, і вони привнесуть суттєву користь:

- Домовтесь про спільні шаблони
- Визначте послідовність дій
- Переконайтесь що стандарти і процеси використовуються
- Проводьте аналіз виконаних проектів
- Аналізуйте і навчайтесь використовуючи дані дефектів
- Використовуйте, те що вивчили

Розглянемо ці кроки детально.

Використання шаблонів

Чи дійсно необхідно п'ятдесят способів іменувати змінні? Отримання згоди групи розробників на використання шаблонів або моделей (стандартів) зробити легше, ніж це звучить. У кожного розробника є свій улюблений спосіб виконувати завдання, і більшість розробників вітають можливість обговорити стандарти, які вони використовують.

Загальні шаблони забезпечують членів команди важливою основою для співпраці. Коли кожен підходить до певного завдання різним способом, співпраця в кращому разі утруднена. Часто розробник боїться попросити допомоги іншої людини, тому що вона може не погодитися з його підходом. А коли співпраці немає, такі відмінності в підходах можуть перешкоджати загальному розумінню і накопиченню знань і досвіду.

Дії з контролю якості (Quality Control), такі як рецензії і тестування, могли б бути краще сфокусированы і продуктивніші, якщо виріб був би побудований, використовуючи узгоджені шаблони. Без них, рецензенти і тестувальники повинні пробувати знайти баги в чому завгодно, що розробник міг зробити. Такий неорганізований підхід до контролю якості вимагає великих зусиль і закінчується меншим об’ємом і нижчим результатом виявлення дефектів.

Шаблони також можуть підштовхнути кращу технічну роботу. Розробник, який робить роботу своїм власним способом, може легко пропустити важливі кроки або проглянути

важливу інформацію. З наявним стандартом для роботи, немає ніяких питань щодо того, як повинна виглядати закінчена робота, і що вона повинна в себе включати.

Ви повинні розглянути **стандарти для планів, специфікацій, коду, інтерфейсів користувача, документів, керівництва (повчальних матеріалів), і будь-яких інших компонент роботи**, тому що згода про те, як проект повинен бути зроблений, може допомогти гарантувати його якість. Але разом із стандартами, ви повинні ідентифікувати стани, в яких вони можуть використовуватися і забезпечити керівництвом для їх відтворення, коли необхідно. Будь-який стандарт, з яким ви погоджуєтеся, повинен допомогти вам виконувати роботу якнайкращим способом і не повинен заважати їй.

Створення інструкцій

Зауважте, що заголовок не говорить, що ви повинні використовувати інструкції або процеси. Це тому що кожен використовує процеси для речей, які робить регулярно. Ваш відділ вже має інструкції по створенню програмного забезпечення. Ось два головні питання, які ви повинні запитати самі себе щодо цих інструкцій:

1. Чи відповідають вони вашим потребам?
2. Чи виконуєте ви їх послідовно у відповідних середовищах

Перше питання зосереджено безпосередньо на якості інструкцій. Чи підходять вони для своїх цілей?

Залежно від мети, ви можете поставити наступні питання:

Підтримують і чи заохочують вони рівень співпраці, якої ви потребуєте?

Чи просувають вони достатній обмін між командою розробників і клієнтами?

Чи підтримують вони використання кращих технічних стандартів?

Чи допомагають вони вам досягти ваших цілей якості?

Наприклад, процеси можуть йти по шляху співпраці, технічної переваги, або відносин зацікавлених сторін, або деяким іншим способом будуть не в змозі виконувати потреби команди. Людина, яка говорить “Я ніколи не зустрічав процес, який би мені подобався”, ймовірно, використовувала багато хороших процесів, але не розуміла їх.

Тоді як високоякісні процеси не завжди невидимі, вони примушують виконувати роботу гладше. Вони сприяють перевазі при забезпеченні гнучкості, щоб пристосуватися до унікальних вимог кожного проекту. Коротше кажучи, вони підтримують потреби ваших проектів.

Друге питання вказує на якість вашого виконання цих інструкцій. Якщо ви не доречно і не послідовно робите речі, з якими ви погодилися, то вигоди від хороших процесів, ймовірно, будуть втрачені. Послідовне використання високоякісних процесів - питання упевненості в тому, що кожна людина знає, коли і як слідувати їм і строге дотримання цих інструкцій - очікуваний результат для команди. “

Використання стандартів і процесів

Щоб отримувати повну вигоду із стандартів і процесів, які ви встановлюєте, необхідно безперервно перевіряти, що ви отримуєте передбачуваний результат

Модель CMMI (Capability Maturity Model Integration) робить це через перевірку процесів. (CMMI класифікує ці перевірки як забезпечення якості, тому що вони перевіряють процес більше, ніж продукт.) Гнучкі технології програмування (наприклад, Екстремальне Програмування - Extreme Programming або XP) використовують для цього метод коучинга (coaching). Незалежно від того, як ця перевірка зроблена або як вона називається, вона закінчується вигодами якості.

Якщо ви виявляєте випадок, де прийнятий стандарт або процес ігнорувалися, це заслуговує дослідження і вирішення, тому що для цього може бути безліч причин. Наприклад:

Людина просто забула слідувати за стандартом або процесом. Нагадаєте йому.

Чоловік не розумів стандарт або процес або не знав, як використовувати його. Поліпшите ваш процес передачі інформації або повчальні методи.

Стандарт або процес не підходив для даної специфічної роботи. Перегляньете керівництво, що пристосовують для певної мети, або альтернативи.

Стандарт або процес був неефективним або дуже громіздким для ситуації. Спростіть його так, щоб він відповідав вимозі.

Кожне "порушення" стандарту або процесу – це можливість вивчити і покращувати його так, щоб він краще відповідав потребам команди.

Аналіз виконаних проектів

Вивчені уроки (Lessons Learned), пост-**програми** (Post Programs or Post Project Analysis) – ось деякі з найбільш могутніх інструментів для активного поліпшення якості вашої роботи. Подальший аналіз – це момент, щоб озирнутися назад на проект і спробувати навчитися на власному досвіді. Ключові питання: “Що пішло добре, і як я можу відтворити це в майбутньому?” і “Що пішло не так як треба, і як я можу запобігати цьому в майбутньому?”

Хоча подальші аналізи широко визнані як “краща практика” (Best Practise) ми переконалися в тому, що вони досить рідкісні. Дві найбільш часто висловлювані причини для цього: “Важко зібрати всіх разом для обговорення і подальшого анилізу проекту” і ” Ми робили це, але це ніколи не мало ніякого ефекту”.

Перша причина виникає, тому що збори по подальших аналізах проводяться в кінці проектів. Багато хто з учасників проекту перейшов, і ті, хто залишилися, зайняті справою випуску продукту і початковою підтримкою. "Agile" методи рекомендують легке вирішення цієї проблеми: Не робіть одиничний подальший аналіз в кінці проекту; робіть подальші аналізи регулярно на протязі всього проекту. Вигоди від цього включають:

- Учасники проекту доступні, тому що вони все ще залучені в проект.
- Проводити аналіз разів на місяць або в інший проміжок часу буде легко, тому що це займатиме годину або два замість дня або більше.
- Досвід членів команди все ще свіжий в їх розумах, і будуть включений в обговорення.
- Найбільш важливе те, що ви можете застосовувати подальші аналізи до робіт, що залишилися, за проектом.

Друга причина, чому подальші аналізи мають тенденцію бути рідкісними – це те, що ви часто збираєте багато цікавої практичної інформації, але у вас немає можливості використовувати цю інформацію в роботі над майбутніми проектами.

Використання Даних Дефекту

Ваша база дефектів – це запис всіх знайдених упущень в якості, аналіз яких необхідний для поліпшення якості ваших майбутніх проектів. Якщо ви не документуєте дефекти у ваших виробах, то сьогодні - хороший час, щоб почати. Якщо ви збираєте деякі дані про дефекти (наприклад, тільки після випуску або тільки на “великих” або тільки на пізніх стадіях розробки), то ви можете захотіти розширити те, що ви збираєте.

Дані про дефекти, які є корисними для удосконалення якості, включають:

- Що було неправильне? Це - не ознака, але проблема, яка повинна була бути виправлена.
- Коли проблема була створена? Чия діяльність під час розробки була джерелом проблеми? Чи було це вимогою до розробки? Дизайном системи? Кодування? Тестування?
- Коли проблема була локалізована? Вона могла бути не виправлена відразу ж, але що нас дійсно хвилює – це те, як довго дефект існував, поки його не виявили.
- Як проблема була знайдена? Це може стати практикою, яку ви здійснюватимете постійно.
- Коли проблема повинна бути виявлена? Чи є процес контролю якості (Quality Control - QC), який не такий ефективний, як це повинно бути?

Скільки це коштувало? Легко виконати загальний підрахунок. Переконаєтеся, що обдумали всю перевірку і доопрацювання, яке ви повинні зробити, включаючи повторне проектування, повторне кодування, повторну збірку, відновлення, перероблення тестів, повторне тестування, повторний випуск, випуск латочок, управління повідомлення про дефект, статусом повідомлення і так далі

Яка це була проблема? Коли у вас більше, ніж декілька десятків дефектів, тоді розподіл дефектів по категоріях полегшує аналіз і вивчення.

Коли ви аналізуєте ваші дані про дефекти, приділяйте більше уваги тим, які трапляються послідовно і які мають високі витрати на виправлення. Вони - ті, яких ви хочете уникнути в майбутньому (або, принаймні, виявити раніше в процесі розробки), тому що це буде найбільшим удосконаленням якості.

Використовуйте отримані знання

Багато **дій із забезпечення якості** (QA), забезпечать вас величезним резервом інформації щодо ваших можливостей для поліпшення якості того, що ви будуєте. Але одна ця інформація не гарантуватиме якість будь-якого майбутнього продукту. Ви повинні використовувати певні дії, щоб застосувати ці знання. Наприклад, якщо ваш процес проектування дуже незручний, щоб використовуватися ефективно для певних типів проектів, то ви повинні розробити альтернативний процес і використовувати його для всіх майбутніх проектів даного типу. На регулярній підставі (наприклад, щорічно, щомісячно, щодня), зупиняйтеся і визначайте ваші можливості для удосконалення, потім вносите зміни до ваших стандартів, процесів, і методів, щоб включити ці удосконалення. Якщо це спеціально не заплановано і зроблено, чиїм-небудь обов'язком, ця діяльність не буде реалізована.

Кінець кінцем, **ваш процес проектних нововведень повинен включати кроки, щоб вчитися від попередніх проектів.** Дивитеся на вашу базу даних подальших аналізів і ваші звіти про дефекти, щоб визначити, як цей проект повинен працювати на відміну від минулих проектів. Які дії ви можете використовувати цього разу, щоб забезпечити (гарантувати) кращу якість, чим було досягнуто раніше? Знову, це повинне бути явне завдання протягом проектних нововведень, або це пропускатиметься під натиском появи кожного нового проекту.

Рекомендації по забезпеченню якості

Ви зобов'язані пам'ятати, що **забезпечення якості** - це не тестування, і воно не направлене на зміну тільки процесу тестування. Ви повинні розуміти, що це поняття охоплює всі технологічні етапи розробки, випуску і експлуатації програмного забезпечення інформаційних систем, що робляться на різних стадіях життєвого циклу, для забезпечення якості продукту, що випускається. Тому будь-яка невдала зміна може вплинути на якість кінцевого продукту. Виходячи з цього я пропоную Вам наступний **план проведення заходів щодо забезпечення якості:**

1. Досліджуйте всі процеси, з якими працює організація. Складіть список **всіх стандартів і процесів, інструкцій, шаблонів** і відзначте ті, які потрібно допрацювати або змінити. Так само напишіть список того, що потрібно додати в існуючий процес. Кожна планована зміна повинна бути ретельно продумане і обгрунтоване. (дивитеся пункт зміна процесів і процедур для отримання докладнішої інформації)
2. Влаштуйте зустріч з керівниками тих груп, де необхідне щось змінити. Це **дуже важливо** в спокійній обстановці (без нервів) **роз'яснити** те, **що треба змінити, що треба прибрати і що треба дописати, а головне навіщо це треба.** Не бійтеся визнати, що якісь зміни, заплановані вами, можна переглянути або відмовитися від них, взагалі, у випадку якщо доводи співробітників компанії переважують ваші (можливо ви не дуже добре обгрунтували вашу точку зору або вашу пропозицію дійсно не дасть необхідної вигоди). Якщо ви це зробите, то можете вважати, що ви справилися з початковою стадією своєї роботи.
3. Всі зміни проводите якомога акуратніше, оскільки будь-який ніяковий рух може спричинити необоротні наслідки. Люди властиві звикати до чого або, тому будь-яка зміна може доставити масу незручностей. Тому не починайте міняти відразу все, вводите нове поступово, при необхідності роз'яснюючи співробітникам компанії навіщо це треба.
4. Перевіряйте, що запропоновані вами зміни виконуються і приносять належний результат. Вимагайте від керівників груп чіткого проходження процесам, інструкціям, а також використання шаблонів.
5. Не зупиняйтеся на досягнутому. Продовжуйте пошук того, що можна поліпшити для створення якіснішого продукту, на підставі нових вже упроваджених процесів, стандартів, інструкцій, шаблонів.

Метрики по забезпеченню якості

Згідно міжнародному стандарту ISO 14598:
Метрика - це кількісний масштаб і метод, який може використовуватися для вимірювання.
Від себе додамо, що введення і використання метрик необхідне для поліпшення контролю над процесом розробки, а зокрема над процесом тестування, який ми і розглядатимемо далі.
Мета контролю тестування полягає в отриманні зворотного зв'язку і візуалізації процесу тестування. Необхідну для контролю інформацію збирають (як в ручну так і автоматично) і використовують для оцінки стану і ухвалення рішень, таких як покриття (наприклад, покриття вимог або коди тестами) або критерії виходу (наприклад, критерії закінчення тестування). Метрики, так само можуть бути використані для оцінки прогресу виконання запланованих робіт і освоєнню бюджету

Створення, використання і аналіз метрик

На наш погляд, для більшої наочності має сенс згрупувати метрики по типах суті, що бере участь в забезпеченні якості і тестуванні програмного забезпечення, а саме:

1. Метрики по тестових випадках (Test Cases)
2. Метрики по багам / дефектам
3. Метрики по завданнях

Хочемо відзначити, що метрики "Open/Closed Bugs", "Bugs by Severity" і "Bugs by Priority" добре візуалізують ступінь наближення продукту до досягнення критеріїв якості по багам. Маючи вимоги до кількості відкритих багів, після кожної ітерації тестування ми порівнюємо їх з реальними даними, тим самим бачивши місця, де нам потрібно додати, для швидкого досягнення мети.

Метрики "Reopened/Closed Bugs" і "Rejected/Opened Bugs" направлені на відстежування роботи окремих учасників груп розробки і тестування.

Приклад перший: Допустимо, ми маємо ситуацію, коли кількість перевідкритих після відлагодження багів не зменшується або навіть росте. Це є сигналом до того, що необхідно провести аналіз причин, оскільки подібна ситуація може показати, що:

1. Вимоги до функції можна трактувати по різному
2. Тестувальник не точно описав проблему
3. Неякісне поверхневе вирішення проблеми (фікс бага)

Другий приклад покаже для чого необхідна метрика "Rejected/Opened Bugs": Ми спостерігаємо, що відсоток відхилених (Rejected) багів дуже великий. Це може означати:

1. Вимоги до функції можна трактувати по різному
2. Тестувальник не точно описав проблему
3. Розробник не бажає виправляти допущену ним помилку або не вважає, що це насправді помилка. (Ця проблема є прямим наслідком 2-ої, такої, що виникла із-за не точного опису)

Всі ці проблеми помітно дестабілізують обстановку на проекті. Тому, при їх виникненні, рекомендується провести коротку бесіду з керівниками проектних груп, щоб надалі зменшити кількість відхилених дефектів.

Метрики по завданнях можуть бути різні, ми привели лише 2 з них. Так само цікава може бути метрика за часом виконання завдань і багато інших.

Відзначимо, що наявність необхідних метрик і графіків, що відображають зміну стану проекту за час, дозволить вам поліпшити не тільки процес тестування, але і розробки в цілому, а також полегшить процедуру проведення аналізу виконаного проекту, що дозволить надалі не допускати минулих помилок.

2.4. Стандарти в інженерії якості.

Перелік стандартів програмної інженерії

1. ISO 9000 Системи керування якістю — Основні положення і словник.
2. ISO 9000-1 Стандарти з керування якістю та забезпечення якості. Ч.1. Настанови щодо вибору та застосування.

3. ISO 9000-2 Стандарти з керування якістю та забезпечення якості. Ч.2. Настанови щодо застосування ISO 9001, ISO 9002, ISO 9003.
4. ISO 9000-3 Стандарти з керування якістю та забезпечення якості – Ч.3. Настанови щодо застосування ISO 9001 під час розроблення, постачання та супроводження програмного забезпечення.
5. ISO 9000-4 Стандарти з керування якістю та забезпечення якості – Ч.4. Настанови щодо керування програмою надійності.
6. ISO 9001-2001 Системи керування якістю. Вимоги.
7. ISO 9003 Система якості. Модель забезпечення якості в процесі контролю готової продукції та її випробуваннях.
8. ISO/IEC 9126-1:2001 Програмна інженерія. Якість продукту.Ч.1. Модель якості.
9. ISO/IEC TR 9126-2:2003 Програмна інженерія. Якість продукту.Ч.2. Зовнішні метрики
10. ISO/IEC TR 9126-3:2003 Програмна інженерія. Якість продукту.Ч.3. Внутрішні метрики.
11. ISO/IEC TR 9126-4:2004 Програмна інженерія. Якість продукту.Ч.4. Метрики якості при використанні.
12. ISO/IEC 14598-2:2000 Програмна інженерія. Оцінювання програмного продукту. Ч.2. Планування та керування.
13. ISO/IEC 14598-3:2000 Програмна інженерія. Оцінювання програмного продукту. Ч.3. Процес для розробників.
14. ISO/IEC 14598-4:1999 Програмна інженерія. Оцінювання продукту. Ч.4. Процес для замовників.
15. ISO/IEC 14598-6:2001 Програмна інженерія. – Оцінка продукту – Ч.6. Документація модулів оцінювання.
16. ISO/IEC 19761:2003 Програмна інженерія. COSMIC-FFP. Метод вимірювання об’єму функціональних можливостей.
17. ISO/IEC 25000:2005 Програмна інженерія. Оцінювання і вимоги до якості програмного продукту(SQuaRE). Настанова щодо оцінювання і вимоги до якості програмного продукту.
18. ISO/IEC 25001:2007 Програмна інженерія. Оцінювання і вимоги до якості програмного продукту(SQuaRE). Планування та керування.
19. ISO/IEC 25020:2007 Програмна інженерія. Оцінювання і вимоги до якості програмного продукту(SQuaRE). Настанова та еталонна модель вимірювання.
20. ISO/IEC TR 25021:2007 Програмна інженерія. Оцінювання і вимоги до якості програмного продукту(ОВЯПП - SQuaRE). Елементи вимірювання якості.
21. ISO/IEC 25030:2007 Програмна інженерія. Оцінювання і вимоги до якості програмного продукту(SQuaRE). Вимоги якості.
22. ISO/IEC 25051:2006 Програмна інженерія. Оцінювання і вимоги до якості програмного продукту(SQuaRE). Вимоги до якості готових комерційних продуктів програмних засобів та інструкції по тестуванню.
23. ISO/IEC 25062:2006 Програмна інженерія. Оцінювання і вимоги до якості програмного продукту(SQuaRE). Загальний промисловий формат, який використовується при тестуванні звітів.
24. ДСТУ ISO/IEC TR 9126-2 Програмна інженерія. Якість продукту.Ч.2 Зовнішні метрики.
25. ДСТУ ISO/IEC TR 9126-3 Програмна інженерія. Якість продукту.Ч.3 Внутрішні метрики.
26. ДСТУ ISO/IEC TR 9126-4 Програмна інженерія. Якість продукту.Ч.4 Метрики якості при використанні.
27. ДСТУ ISO/IEC 14598-1 Інформаційні технології. Оцінювання програмного продукту. Ч.1 Загальний огляд.
28. ДСТУ ISO/IEC 14598-2 Програмна інженерія. Оцінювання програмного продукту. Ч.2. Планування та керування.

29. ДСТУ ISO/IEC 14598-3 Програмна інженерія. Оцінювання програмного продукту. Ч.3. Процес для розробників.
30. ДСТУ ISO/IEC 14598-4 Програмна інженерія. Оцінювання продукту. Частина 4. Процес для замовників.
31. ДСТУ ISO/IEC 14598-6 Програмна інженерія. – Оцінка продукту – Частина 6. Документація модулів оцінювання.

2.5.Метрики якості

Метрика як основа вимірювання. Класифікація мір якості. Класифікація метрик якості. Ключові метрики для контролю розробки програмного забезпечення. Узагальнена модель якості. Метрики в узагальненій моделі якості. Ієрархічні та не ієрархічні моделі якості програмних систем. Графічні моделі якості. Методи оцінки значень показників якості. Основні поняття в проблематиці надійності програмних систем. Класифікація моделей оцінки надійності. Побудова і застосування метрик і моделей якості. Специфікація вимог до якості.

На сьогодні у програмній інженерії ще не сформувалася остаточно система метрик. Діють різні підходи до визначення їхнього набору й методів вимірювання [11–15]. Система вимірювання містить у собі метрики й моделі вимірювань, які використовуються для кількісної оцінки якості ПС. При визначенні вимог до ПС задаються відповідні ним зовнішні характеристики і їхні атрибути (характеристики), що визначають різні аспекти керування продуктом у певному середовищі. Для набору характеристик якості ПС, наведених у вимогах, визначаються відповідні метрики, моделі їхньої оцінки й діапазон значень мір для вимірювання окремих атрибутів якості.

Відповідно до стандарту [1] метрики визначаються за моделями виміру атрибутів ПС на всіх процесах ЖЦ (проміжна, внутрішня метрика) і особливо на процесі тестування або функціонування (зовнішні метрики) продукту. Наведемо класифікацію метрик ПС, правил для проведення метричного аналізу й процесу їхнього виміру.

Типи метрик. Існує три типи метрик:

- метрики програмного продукту, які використовуються для вимірювання його характеристик – властивостей;
- метрики процесу, які використовуються для вимірювання властивості процесу ЖЦ створення продукту;
- метрики використання.

Метрики програмного продукту містять у собі:

- зовнішні метрики, що визначають властивості продукту, видимі користувачеві;
- внутрішні метрики, що визначають властивості, видимі тільки команді розробників.

Зовнішні метрики продукту – це метрики:

- надійності продукту, які використовують для визначення числа дефектів;
- функціональності, за допомогою яких визначають наявність і правильність реалізації функцій у продукті;
- супроводу, за допомогою яких вимірюють ресурси продукту (швидкість, пам'ять, середовище);
- застосування продукту, які сприяють визначенню ступеня доступності для вивчення й використання;
- вартості створеного продукту.

Внутрішні метрики продукту вміщують:

- метрики розміру, необхідні для вимірювання продукту за допомогою його внутрішніх характеристик;
- метрики складності, необхідні для визначення складності продукту;
- метрики стилю, які використовуються для визначення підходів і технологій створення окремих компонентів продукту і його документів.

Внутрішні метрики дозволяють визначити продуктивність продукту і є релевантними відносно зовнішніх метрик.

Зовнішні й внутрішні метрики задають на процесі формування вимог до ПС і є предметом планування й керування досягненням якості кінцевого програмного продукту.

Метрики продукту часто описуються комплексом моделей для встановлення різних властивостей, значень моделі якості або прогнозування. Вимірювання виконують, як правило, після калібрування метрик на ранніх процесах проекту. Загальна міра – ступінь трасування, що визначають числом трас, які простежуються за моделями сценаріїв типу UML й оцінкою кількості:

– вимог;

– сценаріїв і дійових осіб;

– об'єктів, вміщених у сценарій, і локалізація вимог до кожного сценарію;

– параметрів й операцій об'єкта й ін.

Стандарт ISO/IEC 9126–2 визначає такі типи мір:

– міра розміру ПС в різних одиницях вимірювання (число функцій, рядків у програмі, розмір дискової пам'яті й ін.);

– міра часу (функціонування системи, виконання компонента й ін.);

– міра зусиль (продуктивність праці, трудомісткість й ін.);

– міра обліку (кількість помилок, число відмов, відповідей системи та ін.).

Спеціальною мірою може бути рівень використання повторних компонентів, яку вимірюють як відношення розміру продукту, виготовленого з готових компонентів, до розміру системи в цілому. Така міра використовується також при визначенні вартості і якості ПС. Приклади метрик:

- загальне число об'єктів і число повторно використовуваних;
- загальне число операцій, повторно використовуваних і нових операцій;
- число класів, що успадковують специфічні операції;
- число класів, від яких залежить певний клас;
- число користувачів класу або операцій та ін.

При оцінці загальної кількості певних величин часто використовують середньостатистичні метрики (середнє число операцій у класі, класу нащадків або операцій класу й ін.).

Як правило, міри є суб'єктивними й залежать від знань експертів, що виконують кількісні оцінки атрибутів компонентів програмного продукту.

Прикладом широко використовуваних зовнішніх метрик програм є метрики Холстеда – це характеристики програм, виявлених на основі статичної структури програми конкретною мовою програмування: число входжень операндів й операторів, що найчастіше зустрічаються; довжина опису програми як сума числа входжень всіх операндів й операторів та ін.

На основі цих атрибутів можна обчислити час програмування, рівень програми (структурованість та якість) і мови програмування (абстракції засобів мови й орієнтація на проблему) та ін.

Як метрику процесу можна використовувати час розробки, число помилок, знайдених на процесі тестування та ін. Частіше застосовують такі метрики процесу:

- загальний час розробки й час окремо для кожної стадії;
- час модифікації моделей;
- час виконання робіт на процесі;
- число знайдених помилок при інспектуванні;
- вартість перевірки якості;
- вартість процесу розробки.

Метрики використання призначено для вимірювання ступеня задоволення потреб користувача для розв’язання задач. Вони допомагають оцінити не властивості самої програми, а результати її експлуатації – експлуатаційну якість. Як приклад – точність і повнота реалізації завдань користувача, а також витрачені ресурси (трудовитрати,

продуктивність та ін.) на ефективне розв’язання задач користувача. Оцінка вимог користувача виконується за допомогою зовнішніх метрик.

Модель якості програмних систем

Якість ПС – це відносне поняття, що має сенс тільки з урахуванням реальних умов його застосування, тому вимоги до якості висуваються відповідно до умов та конкретної сфери їхнього використання. Якість характеризується трьома аспектами: якість програмного продукту, якість процесів ЖЦ й якість супроводу або впровадження (рис. 9.1).

Рис. 2.1. Основні аспекти якості ПС

Аспект, пов'язаний із процесами ЖЦ, визначає ступінь формалізації, вірогідності процесів ЖЦ з розроблення ПС, а також верифікацію й валідацію проміжних і кінцевих результатів на цих процесах. Пошук й усунення помилок у готовому ПС проводиться за допомогою методів тестування, які зменшують кількість помилок і підвищують якість цього продукту.

Якість продукту досягається процедурами контролю проміжних продуктів під час процесів ЖЦ, перевіркою їх на досягнення необхідної якості, а також методами супроводу продукту. Ефект від впровадження ПС великою мірою залежить від знань обслуговуючого персоналу функцій продукту й правил їхнього виконання.

Модель якості програмного забезпечення (рис. 9.2) має чотири рівні подання.

Перший рівень подання відповідає визначенню характеристик (показників) якості ПС, кожна з яких відображає окреме уявлення користувача про якість.

Відповідно до стандарту [1–4] у модель якості входить шість характеристик або шість показників якості:

1) функціональність (functionality);
2) надійність (realibility);
3) зручність (usability);
4) ефективність (efficiency);
5) супровід (maitainnability);
6) мобільність (portability).

Другому рівню подання відповідають атрибути для кожної характеристики якості, які деталізують різні аспекти конкретної характеристики. Набір атрибутів характеристик якості використовується для оцінки якості.

Третій рівень подання призначено для виміру якості за допомогою метрик, кожна з яких відповідно до стандарту [1] визначається як комбінація методу виміру атрибута й шкали виміру значень атрибутів. Для оцінки атрибутів якості на процесах ЖЦ (при перегляді документації, програм і результатів тестування програм) використовуються метрики із заданою цінною вагою для нівелювання результатів метричного аналізу сукупних атрибутів конкретного показника і якості в цілому. Атрибут якості визначається за допомогою однієї або декількох методик оцінки на процесах ЖЦ і на завершальному процесі розроблення ПС.

Четвертий рівень подання – це оцінний елемент метрики (вага), що використовується для оцінки кількісного або якісного значення окремого атрибута показника ПС. Залежно від призначення, особливостей та умов супроводу вибираються найважливі характеристики якості та їхні атрибути.

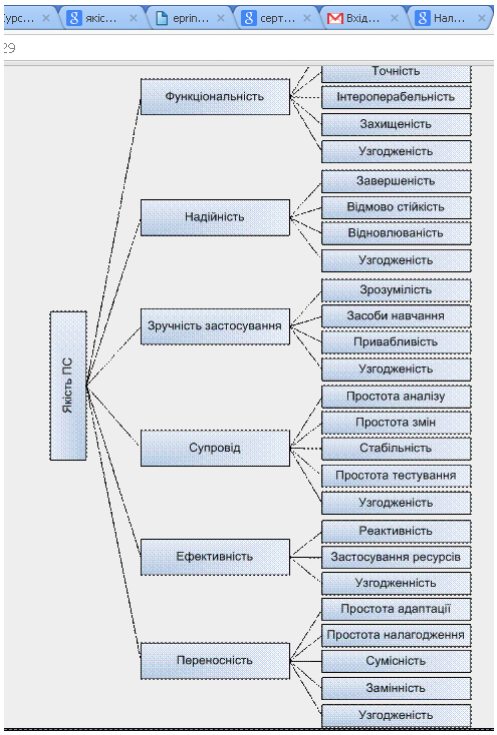

Рис. 2.2. Модель характеристик якості

Вибрані атрибути та їхні пріоритети відображаються у вимогах на розробку систем або використовуються відповідні пріоритети еталона класу ПС, до якого це ПС належить.

2.6.Сертифікація програмного продукту

Під *сертифікацією програмного продукту* розуміють процес, що здійснюється третьою стороною для засвідчення його спеціальним знаком або посвідченням ідентифікованої програмної продукції (процесу або послуги) на відповідність конкретному стандарту, технічним умовам або вимогам.

Сертифікат на програмну продукцію свідчить про відповідність перевірених показників якості цієї продукції певним вимогам. В умовах ринкових відносин наявність такого сертифікату підвищує конкурентоспроможність, є засобом завоювання ринку й захисту споживачів від недоброякісної продукції.

Спеціальним декретом Кабінету Міністрів України «Про стандартизацію й сертифікацію» передбачається два види сертифікації створюваної продукції: обов'язкова й добровільна.

Обов'язкова сертифікація продукту орієнтується на проведення в Державній системі сертифікації Укрсепро перевірок відповідності реальних властивостей сертифікованої продукції вимогам, певним державним нормативним документам. До обов'язкової сертифікації віднесено потенційно небезпечні й шкідливі продукти, вироби, процеси. У цьому переліку не зазначено програмну продукцію, хоча помилки в ній можуть призвести до небезпечних наслідків як для безпеки людей, так і для економіки. Прикладом небезпечних наслідків можуть служити аварії при запуску космічних кораблів «Челенджер» (США, 1995) і «Зеніт-2» (СНД, 1998), причиною яких стали помилки в програмах керування польотом.

У зв'язку із цим ясно, що сертифікація програмної продукції як механізму керування якістю, забезпечення її безпеки й конкурентоспроможності вітчизняних програмних продуктів, захисту користувачів від недоброякісної продукції, необхідна. Проте багато організацій, що створюють програмні продукти, не вживають заходів з забезпечення їхньої якості й сертифікації. Це пояснюється низкою причин:

– небажанням піддавати програмні продукти сертифікації, тому що це вимагає додаткових ресурсів;

– нерозумінням замовника ПС переваг сертифікованого продукту;

– відсутністю в організаціях систем забезпечення якості та ін.;

– відсутністю ринку вітчизняної програмної продукції.

Системи забезпечення якості ПС, нормативно-методичні документи, що визначають найраціональні й ефективні процеси й процедури його реалізації, а також системи сертифікації програмних продуктів, спрямовано на розв'язання таких завдань:

1) створення нормативної бази інженерії якості ПС, що відповідає вимогам міжнародних і державних стандартів;

2) розроблення типових елементів систем забезпечення якості в організаціях, що розробляють програмні продукти;

3) опанування й удосконалювання методів оцінки якості продуктів і процесів їхнього виробництва;

4) створення нормативно-методичної й інструментальної бази системи сертифікації програмних продуктів.

До цих завдань додаються подання оцінки відповідно до моделі CMM [38] зрілості організації й процесів виробництва програмних продуктів.

Висновок. Представлено результати досліджень проблематики якості ПС, що містить у собі методи інженерії якості, метрики, оцінки атрибутів показників якості і якості в цілому. Зазначається, що на оцінку якості ПС впливають методи інженерії вимог до ПС і методи, що гарантують досягнення заданих характеристик на ранніх процесах ЖЦ. Розглянуто моделі надійності, подано їхню класифікацію, наведено моделі марковського й пуассонівського типів, основані на кількості помилок й інтенсивності відмов при тестування ПС.

Контрольні питання й завдання

1. Визначте поняття якість ПС і рівні моделі якості ПС.
2. Визначте характеристики якості ПС і їхнє призначення.
3. Які методи визначають показники якості?
4. Визначте метрики програмного продукту і їхні складові.
5. Які існують стандарти з якості ПС?
6. Назвіть основні цілі й завдання системи керування якістю.
7. Визначте процеси досягнення надійності на ЖЦ.
8. Що таке сертифікація програмного продукту?

ТЕМА 3. ПРОЦЕСИ УПРАВЛІННЯ ЯКІСТЮ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Верифікація і валідація програм

Верифікація і валідація – це методи аналізу, перевірки специфікацій і правильності виконання програм відповідно до заданих вимог і формального опису програми.

Метод верифікації допомагає зробити висновок про коректність створеної програмної системи при її проектуванні і після завершення її розроблення. Валідація дозволяє встановити здійснимість заданих вимог шляхом їх перегляду, інспекції і оцінки результатів проектування на процесах ЖЦ для підтвердження того, що здійснюється коректна реалізація вимог, дотримання заданих умов і обмежень до системи. Верифікація і валідація забезпечують перевірку повноти, несуперечності і однозначності специфікації і правильності виконання функцій системи.

Верифікації і валідації піддаються:

– компоненти системи, їх інтерфейси (програмні, технічні і інформаційні) і взаємодія об'єктів (протоколи, повідомлення) у розподілених середовищах;

– описи доступу до баз даних, засоби захисту від несанкціонованого доступу до даних різних користувачів;

– документація до системи;

– тести, тестові процедури і вхідні набори даних.

Верифікація і валідація як методи перевірки правильності виконання заданих функцій і відповідності їх вимогам замовника подані в стандарті [7-9] у вигляді самостійних процесів ЖЦ і використовуються, починаючи від етапу аналізу вимог і закінчуючи перевіркою правильності функціонування програмного коду на заключному процесі, а саме, під час тестування.

Для цих процесів визначені цілі, задачі і дії з перевірки правильності створюваного проміжного продукту на процесах ЖЦ. Розглянемо їхнє стандартне подання.

Процес верифікації. Мета процесу – переконатися, що кожен програмний продукт (і/або сервіс) проекту відбиває погоджені вимоги до їхньої реалізації. Цей процес ґрунтується:

– на стратегії і критеріях верифікації всіх робочих програмних продуктів на ЖЦ;

– на виконанні дій з верифікації відповідно до стандарту;

– на усуненні недоліків, виявлених у програмних (робочих, проміжних і кінцевих) продуктах;

– на узгодженні результатів верифікації з замовником.

Процес верифікації може проводитися виконавцем програми або іншим співробітником тієї ж організації, або співробітником іншої організації, наприклад представником замовника. Цей процес містить у собі дії з його впровадження і виконання.

Впровадження процесу полягає у визначенні критичних елементів (процесів і програмних продуктів), що повинні піддаватися верифікації, у виборі виконавця верифікації, інструментальних засобів підтримки процесу верифікації, у складанні плану верифікації і його затвердження. У процесі верифікації виконуються задачі перевірки умов: контракту, процесу, вимог, інтеграції, коду і документації.

Відповідно до плану і вимог замовника перевіряється правильність виконання функцій системи, інтерфейсів і взаємозв'язків компонентів, а також доступ до даних і до засобів захисту.

Процес валідації. Мета процесу – переконатися, що специфічні вимоги для програмного продукту виконано, і здійснюється це за допомогою:

– розробленої стратегії і критеріїв перевірки всіх робочих продуктів;

– обговорених дій з проведення валідації;

– демонстрації відповідності розроблених програмних продуктів вимогам замовника і правилам їхнього використання;

– узгодження із замовником отриманих результатів валідації продукту.

Процес валідації може проводитися самим виконавцем або іншою особою, наприклад, замовником, що здійснює дії з впровадженню і проведенню цього процесу за планом, у якому відбиті елементи і задачі перевірки. При цьому використовуються методи, інструментальні засоби і процедури виконання задач процесу для встановлення відповідності тестових вимог і особливостей використання програмних продуктів проекту на правильність реалізації вимог.

На інших процесах ЖЦ виконуються додаткові дії:

– перевірка і контроль проектних рішень за допомогою методик і процедур перегляду ходу розроблення;

– звернення до CASE-систем, що містять у собі процедури перевірки вимог до продукту;

– перегляди й інспекції проміжних результатів на відповідність вимогам для підтвердження того, що ПС має коректну реалізацію вимог і задовольняє умови виконання системи.

Таким чином, основні задачі процесів верифікації і валідації полягають у тому, щоб *перевірити і підтвердити*, що кінцевий програмний продукт відповідає призначенню і задовольняє вимогам замовника. Ці процеси взаємозалежні і визначаються, як правило, одним загальним терміном «верифікація і валідація» або «Verification and Validation» (V&V).

V&V засновані на плануванні їх як процесів, так і перевірки для найбільш критичних елементів проекту: компонентів, інтерфейсів (програмних, технічних і інформаційних), взаємодій об'єктів (протоколів і повідомлень), передачі даних між компонентами і їхнього захисту, а також створення тестів і тестових процедур.

Після перевірки окремих компонентів системи проводяться їхня інтеграція, повторна верифікація і валідація інтегрованої системи, створюється комплект документації, що відображає правильність виконання вимог за результатами інспекцій і тестування.

3.1.1. Підхід до валідації сценарію вимог

До процесу створення програм належить опис вимог мовою UML за допомогою сценаріїв і діючих виконавців – акторів як зовнішніх сутностей щодо системи [22]. Вимоги потрібно перевіряти до їхньої перебудови у програмні елементи. Сценарій після трансформації – це послідовність взаємодій між одним або декількома акторами і системою, у якій актор виконує мету сценарію при взаємодії з нею. У моделі вимог сценарій задає кілька альтернативних подій, заданих мовою діаграм UML. Вони розділяються на функціональні (системні) і внутрішні, як визначальне поводження системи. На основі опису сценарію вимоги перевіряються шляхом валідації для виявлення помилок у поданні сценарних вимог. Ця перевірка відбувається ітераційною і складається з наступних кроків:

1. Формалізований опис вимог у вигляді сценаріїв;
2. Створення моделі вимог;
3. Створення спеціальних сценаріїв для валідації вимог;
4. Застосування валідаційних сценаріїв у моделі вимог;
5. Оцінювання результатів поводження моделі вимог;
3. Перевірка умов завершення процесу валідації і при виявленні яких-небудь неточностей повторення кроків, починаючи з п. 2.

При виконанні сценаріїв можуть виникнути помилкові ситуації, за яких поведінки системи стає не детермінованим. За цих цілей проводиться контроль покриття сценаріїв у моделі вимог валідаційними сценаріями з метою виявлення помилок або ризиків (рис. 3.3).

Створюється модель помилок, що покриває модель вимог системи з типовими помилками, що використовуються при доведенні сценарієв.

Складова частина валідації вимог за сценаріями – визначення класів еквівалентності вхідних і вихідних даних для валідації і синтезу сценаріїв. Вхідна інформація для синтезу сценаріїв – сценарна модель, що задається мовою взаємодії.

Інформація використовується при генерації додаткових сценаріїв з метою поліпшення процесу валідації, автоматичного синтезу сценаріїв моделі й отримання моделі поведінки системи під керуванням актора.

Модель перевіряється за допомогою тестів і моделі помилок, що в цілому дозволяє знайти неповноту вихідних вимог або суперечності у вимогах.

Автоматичний синтез програми заснований на наступних процедурах:

– валідація вимог шляхом виконання валідаційних сценаріїв;

– додавання перевірених сценаріїв до набору валідаційних сценаріїв і їхнє використання як вхідних даних для синтезу;

– пошук помилок у сценаріях і перевірка різних композицій сценаріїв.

Синтез специфікацій сценаріїв вимог, трансформованих до діаграмам взаємодії, може проводитися в середовищі системи Rational Rose.

3.1.2. Верифікація об'єктних моделей

Верифікація об'єктної моделі (ОМ) ґрунтується на специфікації:

– базових (простих) об'єктів ОМ, атрибутами яких є дані та операції об'єкта – функції над цими даними;

– об'єктів, які вважаються перевіреними, якщо їх операції використовуються як теореми, що застосовуються над підоб'єктами і не виводять їх з множини станів цих об'єктів.

Доведення правильності побудови ОМ передбачає:

– введення додаткових і (або) видалення зайвих атрибутів об'єкта і його інтерфейсів в ОМ, доведення правильності об'єкта ОМ на основі специфікації інтерфейсів і взаємодій з іншими об'єктами;

– доведення правильності завдання типів для атрибутів об'єкта, тобто правильності того, що вибраний тип реалізує операцію, а множина його значень визначається множиною станів об'єкта.

Це доведення є завершальним при перевірці правильності ОМ.

Верифікація інтерфейсів об'єктів ОМ зводиться до доведення правильності передачі типів і кількості даних в параметрах повідомлень про їхні специфікації в мові IDL. Інтерфейс складається з операцій звернення до об'єкта, який посилає дані іншому об'єкту через повідомлення. Для доведення правильності специфікації повідомлення створюється набір тверджень, який доводить, що для будь-якої пари елементів повідомлення, наприклад, *A* і *B*, перехід від *A* до *B* відбувається за один крок. Дія, що виконується в проміжку між *A* і *B*, приводить до *B*. При цьому частина тверджень перевіряє вхідний параметр і його надходження на вхід іншого об'єкта з метою підтвердження його на виході. Якщо доведено, що об'єкт, ініційований повідомленням, формує правильний вихідний результат у вихідному параметрі, то повідомлення вважається правильним.

Верифікація моделі розподіленого застосування виконується на основі специфікації SDL (Specification Description Language), моделі перевірки (Model Checking), індуктивних тверджень, запропонованих Новосибірською школою програмування.

Метод перевірки полягає в редукції системи з нескінченним числом станів до системи із скінченного числа станів, а також у доведенні коректності розподіленого застосування за допомогою індуктивних міркувань і системи переходів скінченного автомата.

Основні підходи до верифікації – аксіоматичний і семантичний шлях Model Checking.

Аксіоматичний (за методом Hoare) підхід міститься в описі програми набором аксіом для завдання станів з використанням теорія логіки.

Семантичний підхід ґрунтується на теорії темпоральної логіки Манна для завдання специфікації програм. Аксіоми використовуються для керування семантикою мови специфікації.

Основними типами даних специфікації в SDL є наперед визначені і сконструйовані типи даних (масив, послідовність і т.д.). У мові описуються формули за допомогою предикатів, булевих операцій, кванторів, змінних і модальностей. Семантика їх визначення

залежить від можливих послідовностей дій (поведінки), що виконуються специфікацією процесу, а також моменту часу його виконання.

Схема специфікації процесу – це опис умов виконання і діаграм процесів. Вона ініціюється посиланням повідомлення із зовнішнього середовища для виконання. Діаграма процесу складається з описів переходів, станів, набору операцій процесу і переходу до наступного стану.

Кожна операція визначає поведінку процесу і спричиняє деяку подію. Логічна формула задає модальність поведінки специфікації і моменти часу. Процес, наданий формальною специфікацією, виконується не детерміновано. Обмін із зовнішнім середовищем відбувається через вхідні і вихідні параметри повідомлень.

Подія. У кожний момент часу виконання процес має деякий стан, який може бути поданий у вигляді знімка, що характеризує деяку подію, яка містить у собі значення змінних, яким відповідають параметри і характеристики станів процесу.

Таким чином, модель перевірки, набір аксіом мовою логіки і твердження про виконання розподілених програм забезпечують процес їхньої верифікації.

3.1.3. Загальні перспективи верифікації програм

Методи формальної верифікації використовувалися для перевірки правильності моделей ПрО, функцій в мові API, безпеки і цілісності БД – у проекті SDV фірми Microsoft і у *міжнародному проекті з формальної верифікації ПС*.

Ідея створення цього проекту належить Т.Хоару і обговорювалася на симпозіумі з верифікованого ПС у лютому 2005г. у Каліфорнії. Потім у жовтні того ж року на конференції IFIP в Цюріху був прийнятий міжнародний проект строком на 15 років з розроблення цілісного автоматизованого набору інструментів для перевірки коректності ПС. У проекті сформульовані такі основні задачі:

– розроблення єдиної теорії створення і аналізу програм;

– побудова всеосяжного інтегрованого набору інструментів верифікації для всіх процесів, включаючи розроблення специфікацій і їх перевірку, генерацію тестових прикладів, уточнення, аналіз і верифікацію програм;

– створення репозитарію формальних специфікацій і верифікованих програмних об'єктів різних видів і типів.

Репозітарій – це сховище правильних програм, специфікацій і інструментів.

Функції репозитарію:

– накопичення верифікованих специфікацій, методів доведення, програмних об'єктів і реалізацій кодів для різних програмних застосувань;

– накопичення всіляких методів верифікації, їх оформлення у вигляді, придатному для пошуку і відбору реалізованої теоретичної концепції для подальшого застосування;

– розроблення стандартних форм для завдання і обміну формальними специфікаціями різних об'єктів, інструментів і готових систем;

– розроблення механізмів взаємодії для перенесення готових верифікованих продуктів з репозитарію в нові розподілені і мережні середовища для їхнього використання в нових ПС.

Даний проект передбачається розвивати протягом 50 років. Відомо, що більш ранні проекти ставили подібні цілі: поліпшення якості ПС, формалізації сервісних моделей, зниження складності за рахунок використання КПВ, створення налагоджувального інструментарію для візуальної діагностики помилок і їх усунення тощо. Проте корінної зміни в програмуванні поки не відбулося. Залучення техніки формальної специфікації програм ще не означає, що в програмі будуть відсутні помилки, оскільки помилки в програмних проектах, в інтерпретації специфікацій МП, у документації поки не можна розпізнати. Реалізація міжнародного проекту з верифікації ПС допоможе вирішити багато з цих питань.

Контрольні запитання

ТЕМА 4. ОСНОВИ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. ВИДИ І РІВНІ ТЕСТУВАННЯ

4.1. **Процес тестування за життєвим циклом**

Наведені типи помилок розподіляються за процесами ЖЦ і їм відповідають такі джерела їхнього виникнення :

– ненавмисне відхилення розробників від робочих стандартів або планів реалізації;

– специфікації функціональних і інтерфейсних вимог виконані без дотримання стандартів розробки, що призводить до порушення функціонування програм;

– організації процесу розробки – недосконале або недостатнє управління керівником проекту ресурсами (людськими, технічними, програмними і т.д.) і питаннями тестування й інтеграції елементів проекту.

Розглянемо процес тестування, виходячи з рекомендацій стандарту ISO/IEC–12207, і наведемо типи помилок, що виявляються під час кожного процесу ЖЦ.

Процес розробки вимог. При визначенні вихідної концепції системи і вихідних вимог до системи виникають помилки аналітиків при специфікації вищого рівня системи і побудові концептуальної моделі предметної області.

Характерними помилками цього процесу є:

– неадекватність специфікації вимогам кінцевих користувачів;

– некоректність специфікації взаємодії ПС із середовищем функціонування або з користувачами;

– невідповідність вимог замовника окремим і загальним властивостям ПС;

– некоректність опису функціональних характеристик;

– незабезпеченість інструментальними засобами всіх аспектів реалізації вимог замовника й ін.

Процес проектування. Помилки при проектуванні компонентів можуть бути наслідком недоліків в описі алгоритмів, логіки керування, структур даних, інтерфейсів, логіки моделювання потоків даних, форматів вводу-виводу та ін. В основі цих помилок лежать дефекти специфікацій заданих аналітиками і недоробки проектувальників. До них належать помилки, пов'язані з:

– погодженістю інтерфейсу користувача із середовищем;

– описом функцій (неадекватність цілей і задач компонентів, що виявляються при перевірці комплексу компонентів);

– визначенням процесу обробки інформації і взаємодії між процесами (результат некоректного визначення взаємозв'язків компонентів і процесів);

– некоректним завданням даних і їхніх структур при описі окремих компонентів і ПС у цілому;

– некоректним описом алгоритмів модулів;

– визначенням умов виникнення можливих помилок у програмі;

– порушенням прийнятих для проекту стандартів і технологій.

Процес кодування. На даному процесі виникають помилки, що є результатом дефектів проектування, помилок програмістів і менеджерів у процесі розроблення і налагодження системи. Причиною помилок є:

– безконтрольність значень вхідних параметрів, індексів масивів, параметрів циклів, вихідних результатів та ін.;

– неправильна обробка нерегулярних ситуацій при аналізі кодів повернення від викликуваних підпрограм, функцій і ін.;

– порушення стандартів кодування (погані коментарі, нераціональне виділення модулів і компонентів та ін.);

– використання одного імені для позначення різних об'єктів або різних імен одного об'єкта, погана мнемоніка імен;

– непогоджене внесення змін у програму різними розробниками та ін.

Процес тестування. На цьому процесі помилки допускаються програмістами і тестувальниками при виконанні технології збирання і тестування, вибору тестових наборів і сценаріїв тестування та ін. Відмови в програмному забезпеченні, викликані такого роду помилками, повинні виявлятися, усуватися і не впливають на статистику помилок компонентів і на програмне забезпечення в цілому.

Процес супроводу. На процесі супроводу виявляються помилки, причиною яких є недоробки і дефекти експлуатаційної документації, недостатні показники кодифікованості й легкості читання, а також некомпетентність осіб, відповідальних за супровід і/або удосконалення ПС. Залежно від сутності внесених змін на цьому процесі можуть виникати практично будь-які помилки, аналогічні раніше перерахованим помилкам на попередніх процесах.

Джерела помилок. Помилки можуть бути виникнути в процесі розроблення проекту, компонентів, коду і документації. Як правило, вони виявляються при виконанні або супроводі програмного забезпечення в найбільш несподіваних і різних її точках.

Причиною появи помилок є – нерозуміння вимог замовника; неточна специфікація вимог у документах проекту та ін. Це приводить до того, що реалізуються деякі функції системи, що будуть працювати не так, як пропонує замовник. У зв'язку з цим проводиться спільне обговорення замовником і розробником деяких деталей вимог для їхнього уточнення.

Команда розробників системи може також змінити мову опису системи. Деякі помилки можуть бути не виявлені (наприклад, неправильно задані індекси або значення змінних цих операторів).

Визначення тесту. Для перевірки правильності програм спеціально розробляються тести і тестові дані. Під *тестом* розуміється деяка програма, призначена для перевірки працездатності іншої програми і виявлення в ній помилкових ситуацій. Тестову перевірку можна провести також шляхом введення в програму, які перевіряється, операторів, які будуть сигналізувати про хід її виконання й отримання результатів.

Тестові дані слугують для перевірки роботи системи і складаються різними способами: генератором тестових даних, проектною групою на основі документів або наявних файлів, користувачем з специфікаціях вимог та ін. Дуже часто розробляються спеціальні форми вхідних документів, у яких відображається процес виконання програми за допомогою тестових даних [23–25, 31].

Створюються тести, що перевіряють:

– повноту функцій;

– погодженість інтерфейсів;

– коректність виконання функцій і правильність функціонування системи в заданих умовах;

– надійність виконання системи;

– захист від збоїв апаратури і не виявлених помилок та ін.

Тестові дані готуються як для перевірки окремих програмних елементів, так і для груп програм або комплексів на різних стадіях процесу розроблення.

Багато типів тестів готуються замовником для перевірки роботи програмної системи. Структура і зміст тестів залежать від виду елемента тестування, яким може бути модуль, компонент, група компонентів, підсистема або система. Деякі тести залежать від мети і

необхідності знати: чи працює система відповідно до її проекту, чи задоволені вимоги і чи бере участь замовник у перевірці роботи тестів тощо.

Залежно від задач, що ставляться перед тестуванням програм, складаються тести перевірки проміжних результатів проектування елементів на процесах ЖЦ, а також створюються тести іспитів остаточного коду системи.

4.2. Тестування програмного забезпечення - основні поняття і визначення

Тестування програмного забезпечення - перевірка відповідності між реальною і очікуваною поведінкою програми, здійснювана на кінцевому наборі тестів, вибраному певним чином. **У ширшому сенсі, тестування** — це одна з техніки контролю якості, що включає активності по плануванню робіт (Test Management), проектуванню тестів (Test Design), виконанню тестування (Test Execution) і аналізу отриманих результатів (Test Analysis).

- **Верифікація** (verification) - це процес оцінки системи або її компонентів з метою визначення чи задовольняють результати поточного етапу розробки умовам, сформованим на початку цього етапу [IEEE]. Тобто чи виконуються наші цілі, терміни, завдання по розробці проекту, визначені на початку поточної фази.
- **Валідація** (validation) - це визначення відповідності ПЗ що розробляється очікуванням і потребам користувачів, вимогам до системи [BS7925-1].

Тест план (Test Plan)

- **Тест план** (Test Plan) - це документ що описує весь об'єм робіт по тестуванню, починаючи з опису об'єкту, стратегії, розкладу, критеріїв почала і закінчення тестування, до необхідного в процесі роботи устаткування, спеціальних знань, а також оцінки рисок з варіантами їх дозволу.

Кожна методологія або процес намагаються нав'язати нам свої формати оформлення планів тестування. Як приклад, шаблони тест планів від RUP (Rational Unified Process) і стандарту IEEE 829:

1. Test Plan Template RUP
2. Test Plan Template IEEE 829

Придивившись уважніше стає ясно, що обидва документи описують одне і теж, але в різній формі. У випадку, якщо вам не підходить стандартний шаблон або ви вирішили придумати свій власний більш відповідний для вас формат документа, то з досвіду можу сказати, що хороший тест план повинен як мінімум відповідати на наступні питання:

1. **Що треба тестувати?**
   - опис об'єкту тестування: системи, додатки, устаткування
2. **Що тестуватимете?**
   - список функцій і опис тестованої системи і її компонент
3. **Як тестуватимете?**
   - стратегія тестування, а саме: види тестування і їх застосування по відношенню до тестованого об'єкту
4. **Коли тестуватимете?**
   - послідовність проведення робіт: підготовка (Test Preparation), тестування (Testing), аналіз результатів (Test Result Analisys) в розрізі запланованих фаз розробки
5. **Критерії почала тестування:**
   - готовність тестової платформи (тестового стенду)
   - закінченість розробки необхідного функціонала
   - наявність всієї необхідної документації
6. **Критерії закінчення тестування:**
   - результати тестування задовольняють критеріям якості продукту
   - вимоги до кількості відкритих багів виконані
   - витримка певного періоду без зміни початкової коди додатку Code Freeze (CF)
   - витримка певного періоду без відкриття нових багів Zero Bug Bounce (ZBB)

Відповівши в своєму тест плані на вищеперелічені питання, можна вважати, що у вас на руках вже є хороша чернетка документа по плануванню тестування. Далі, щоб документ придбав більш менш серйозний вигляд, пропоную доповнити його наступними пунктами:

- Оточення тестованої системи
- Необхідне для тестування устаткування і програмні засоби
- Ризики і їх дозвіл

Найчастіше на практиці доводиться стикатися з наступними видами тест планів:

1. Майстер Тест План (Master Plan or Master Test Plan)
2. Тест План (Test Plan, назвемо його детальний тест план)
3. План Приймальних Випробувань (Product Acceptance Plan) - документ, що описує набір дій, пов'язаних з приймальним тестуванням: стратегія, дата проведення, відповідальні працівники і так далі (Шаблон плану приймально-здавальних випробувань від RUP)

Тест План - це систематичний підхід до тестування систем (як системного забезпечення так і програмного). План зазвичай містить детальний опис того як буде виглядати остаточна послідовність дій (технічний процес).

1. Вступ (Introduction)
Коротко, але зрозуміло і достатньо про продукт
2. Джерельні Документи (Related Documents)
Список джерел інформації
3. Пункти Тестування (Test Items)
Що буде тестуватися
4. Функціональність буде/не буде Тестуватися (Feature to be/not to be Tested)
Функціональність, яка буде/не буде тестуватися
5. Тест Підхід/ Тест Дизайн/ Тест Випадки (Test Approach/Test Design/Test Cases)
Як це буде тестуватися
3. Критерії Успішного/Неуспішного Результату (Pass/Fail Criteria)
Критерії успішного/неуспішного результату
7. Оцінка (Estimations)
Скільки часу необхідно затратити на тестування
8. Середовище (Environment)
У якому середовищі необхідно тестувати
9. Відповідальність (Resource Responsibilities)
Скільки працівників і хто за що відповідальний
10. Ризики і Наслідки (Risks and Contingencies)
Які проблеми можуть виникнути під час тестування і потенційні рішення
11. Графік Тестування (Scheduler)
Як і коли буде відбуватися інформування, відповідальних осіб за проект, про результати тестування
12. Тестові Результати (Test Deliverables)
Що (дані, документи, інше..) буде створено під час тестування і доставлено разом з продуктом

Планування тестування малих проектів

Процес тестування необхідно планувати, навіть якщо це маленький проект на декілька годин. Як мінімум:

- варто скласти список пунктів які необхідно перевірити,
- виписати основні функції які система повинна виконувати,
- не забути про середовище у якому система буде працювати.

Якщо отримуємо готову програмку і відсутність часу на планування, то варто хоча б на картці записати пункти які будемо тестувати. Під час роботи можна зазначати залежності (взаємозалежні функції), що допоможе якісніше і швидше виконати роботу. Можна також виставити пріоритети навпроти кожного пункту чи функціоналу. Пріоритети швидко допомагають зорієнтуватися, як програмістам так і тестерам, яке завдання виконувати першим.

Явна відмінність Майстер Тест Плану від просто Тест Плану в тому, що майстер тест план є більш статичним внаслідок того, що містить в собі високорівневу (High Level) інформацію, яка не схильна до частої зміни в процесі тестування і перегляду вимог. Сам же детальний тест план, який містить конкретнішу інформацію по стратегії, видам тестуванні, розкладу виконання робіт, є "живим" документом, який постійно зазнає зміни, що відображають реальний стан речей на проекті.

У повсякденному житті на проекті може бути один Майстер Тест План і декілька детальних тест планів, що описують окремі модулі одного застосування.

Коли документ пише одна людина, то він виходить "однобоким", тому радимо проводити періодичне рецензування з боку учасників проектної групи, а так само провести процедуру затвердження документа. Як приклад, привожу список учасників проектної групи, затвердження яких я вважаю за необхідне:

- Провідний тестувальник
- Тест менеджер (менеджер за якістю)
- Керівник розробки
- Менеджер Проекту

Кожен з перерахованих учасників проекту, перед твердженням, проведе рецензію і внесе свої коментарі і пропозиції, які допоможуть зробити Ваш тест план повнішим і якіснішим.

Тестовий випадок (Test Case)

Тестовий випадок (Test Case) - це сукупність кроків, конкретних умов і параметрів, необхідних для перевірки реалізації тестованої функції або її частини.

Під тест кейсом розуміється структура вигляду:

Action > Expected Result > Test Result

На просторах інтернету ви зможете знайти дуже багато інформації про структуру тест кейсів, рівні їх деталізації і кількості перевірок в них, я збираюся розповісти про підхід використовуваному мною, і який я хочу запропонувати використовувати вам.

Кожен тест кейс повинен мати 3 частини:

PreConditions — Список дій, які приводять систему до стану придатному для проведення основної перевірки. Або список умов, виконання яких говорить про те, що система знаходиться в придатному для проведення основного тесту стану.

Test Case Description — Список дій, що переводять систему з одного стану в інше, для отримання результату, на підставі якого можна зробити вивід про задоволення реалізації, поставленим вимогам

PostConditions — Список дій, що переводять систему в первинний стан (стан до проведення тесту - initial state)

Примітка: **Post Conditions** не є обов'язковою частиною. Це швидше за все - правило хорошого тону: "насмітив - прибери за собою". Це особливо актуально при автоматизованому тестуванні, коли за один прогін можна наповнити базу даних сотнею або навіть тисячею некоректних документів.

Приклад тест кейса:

do A1, verify B1

do A2, verify B2

do A3, verify B3

У приведеному прикладі кінцева перевірка - B3. Це означає, що саме вона є ключовою. Значить, A1 і A2 - це дії приводять систему в тестопригодное стан. А B1 і B2 - умови того, що система знаходиться в змозі придатному для тестування.

PostConditions в даному прикладі не було описані, але за логікою речей треба виконати кроки, які б повернули систему в первинний стан. (наприклад, видалили створений запис, або відмінили б зміни зроблені в документі)

Тепер відповімо на питання: "Чому дане розбиття зручно використовувати?"

Відповідь: кінцева перевірка одна, тобто у випадку якщо тест провалений (test failed) буде відразу ясно із-за чого. Оскільки якщо провальними виявляться перевірки B1 і/або B2,

то тест кейс буде заблокований (test blocked), через те, що функцію не можливо привести в тестопригодное стан (стан придатний для проведення тестування), але це не означає, що тестована функція не працює.

Деталізація опису тест кейсів

Існує багато різних думок про рівень деталізації при написанні тест кейсів, а також кількості перевірок в одному тест кейсі. Всі вони по своєму правильні. Моя думка, що рівень деталізації тест кейсів повинен бути такий, щоб забезпечувати розумне співвідношення часу проходження до тестового покриття. Тобто до тих пір, поки покриття тестами певного функціонала не міняється, можна зменшувати деталізацію тест кесов.

Приклад тест кейса :

Назва: Перевірка відображення сторінки

Дія: Відкрити сторінку "Вхід в систему"

Перевірка: Перевірте, що сторінка, що відображається, відповідає сторінці на картинці 1 (і прикладаємо зображення сторінки "Вхід в систему")

У прикладі покриття буде однаковим, але час, який буде потрібно для проходження, буде різним. Мені здається, що другий приклад буде навіть наочніший.

4.3. Тест дизайн

На додаток хочеться сказати, що рішення про вигляд тест кейса і деталізації його опису приймає людина, відповідальна за його створення, - Тест Дизайнер або Тест Аналітик, що володіє необхідним досвідом, і який знає техніки тест дизайну У багатьох компаніях ця роль не виділяється окремо, а довіряється звичайним тестувальникам, що у разі недостатньої кваліфікації може привести переписування тест кейсів.

Пропоную вам ознайомитися з коротким описом найбільш поширеної техніки тест дизайну:

- **Еквівалентне Розділення (Equivalence Partitioning - EP).** Як приклад, у вас є діапазон допустимих значень від 1 до 10, ви повинні вибрати одне вірне значення усередині інтервалу, скажімо, 5, і одне невірне значення поза інтервалом - 0.
- **Аналіз Граничних Значень (Boundary Value Analysis - BVA**). Якщо узяти приклад вище, як значення для позитивного тестування виберемо мінімальну і максимальну межі (1 і 10), і значення більше і менше меж (0 і 11). Аналіз Граничний значень може бути застосований до полів, записів, файлів, або до будь-якого роду суті що має обмеження.
- **Причина / Слідство (Cause/Effect - CE**). Це, як правило, введення комбінацій умов (причин), для отримання відповіді від системи (Слідство). Наприклад, ви перевіряєте можливість додавати клієнта, використовуючи певну екранну форму. Для цього вам необхідно буде ввести декілька полів, таких як "Ім'я", "Адреса", "Номер Телефону" а потім, натиснути кнопку "Додати" - ця "Причина". Після натиснення кнопки "Додати", система додає клієнта в базу даних і показує його номер на екрані - це "Слідство".
- **Передбачення помилки (Error Guessing - EG**). Це коли тест аналітик використовує свої знання системи і здібність до інтерпретації специфікації на предмет того, щоб "передбачити" за яких вхідних умов система може видати помилку. Наприклад, специфікація говорить: "користувач повинен ввести код". Тест аналітик, думатиме: "Що, якщо я не введу код?", "Що, якщо я введу неправильний код? ", і так далі. Це і є передбачення помилки.
- **Вичерпне тестування (Exhaustive Testing - ET**) - це крайній випадок. В межах цієї техніки ви повинні перевірити всі можливі комбінації вхідних значень, і в принципі, це повинно знайти всі проблеми. На практиці застосування цього методу не представляється можливим, через величезну кількість вхідних значень.

Висновок — використання різних методів або техніки тест дизайну є цінною стратегією підготовки тестів. Основна перевага полягає в тому, що якщо хто-небудь запитає, чому ви провели той або інший тест, ви зможете без зусиль відповісти і пояснити, обгрунтувавши це використанням тієї або іншої техніки.

4.3. Bug Report

Для того, щоб команда тестування працювала згуртовано і не відволікалася по питаннях оформлення тест кейсів, у всіх повинен бути **єдиний шаблон або підхід до їх написання.** Те, що пропонуємо ми - це структура **PreConditions, Test Case Description, PostConditions,** і вже ваше особисте рішення - користуватися цією структурою або придумати свій "велосипед".

Баг або дефект репорт - це документ, що описує ситуацію або послідовність дій що привела до некоректної роботи об'єкту тестування, з вказівкою причин і очікуваного результату.

Для отримання детальнішої інформації про баг репорт, ми рекомендуємо Вашій увазі наступну інформацію, ознайомившись з якою ви отримаєте вичерпне уявлення про структуру, особливості написання і деяких інших нюансах, необхідних для написання, хороших баг репортів:

- Структура баг репорту
- Важливість і Пріоритет дефекту
- Написання баг репортів

Структура баг репорту

Різні системи менеджменту дефектами, пропонують нам різні поля для заповнення і різні структури опису дефектів. Нижчеприведена таблиця - це спроба показати те, що на підставі отриманого нами досвіду, ми рекомендуємо вам використовувати у вигляді **шаблону баг репорту**.

Шапка	
Короткий опис (Summary)	Короткий опис проблеми, явно вказуючий на причину і тип помилкової ситуації.
Проект (Project)	Назва тестованого проекту
Компонент застосування (Component)	Назва частини або функції тестованого продукту
Номер версії (Version)	Версія на якій була знайдена помилка
Серйозність (Severity)	Найбільш поширена п'ятирівнева система градації серйозності дефекту: S1 Блокуючий (Blocker) S2 Критичний (Critical) S3 Значний (Major) S4 Незначний (Minor) S5 Тривіальний (Trivial)
Пріоритет (Priority)	Пріоритет дефекту: P1 Високий (High) P2 Середній (Medium) P3 Низький (Low)
Статус (Status)	Статус бага. Залежить від використовуваної процедури і життєвого циклу бага (bug workflow and life cycle)
Автор (Author)	Творець баг репорту
Призначений на (Assigned To)	Ім'я співробітника, призначеного на вирішення проблеми
Оточення	

ОС / Сервіс Пак і так далі / Браузера + версія / ...	Інформація про оточення, на якому був знайдений баг: операційна система, сервіс пак, для WEB тестування - ім'я і версія браузера і так далі
...	
Опис	
Кроки відтворення (Steps to Reproduce)	Кроки, по яких можна легко відтворити ситуацію, що привела до помилки.
Фактичний Результат (Result)	Результат, отриманий після проходження кроків до відтворення
Очікуваний результат (Expected Result)	Очікуваний правильний результат
Доповнення	
Прикріплений файл (Attachment)	Файл з балками, скриншот або будь-який інший документ, який може допомогти прояснити причину помилки або вказати на спосіб вирішення проблеми

Важливість і Пріоритет дефекту

Різні системи баг трекінгу пропонують нам різні шляхи опису серйозності і пріоритету баг репорту, незмінним залишається лише сенс, що вкладається ці поля. Всі знають такий баг-трекер, як Atlassian JIRA. У нім, починаючи з якоїсь версії замість одночасного використання полів Severity і Priority, залишили тільки Priority, яке зібрало в собі властивості обидва полів: Originally, JIRA did have both a Priority and a Severity field. The Severity field was removed for a number of reasons... Таким чином, ті хто звик працювати з JIRA не завжди розуміють різницю між цими поняттями, оскільки не мали досвіду їх сумісного використання. Виходячи з особистого досвіду, я наполягаю на розділенні цих понять, а точніше на використанні обидва полів Severity і Priority, оскільки сенс, що вкладається в них, різний:

Серйозність (**Severity**) - це атрибут, що характеризує вплив дефекту на працездатність застосування.

Пріоритет (**Priority**) - це атрибут, вказуючий на черговість виконання завдання або усунення дефекту. Можна сказати, що це інструмент менеджера по плануванню робіт. Чим вище пріоритет, тим швидше потрібно виправити дефект.

Градація Серйозності дефекту (Severity)

S1 Блокуюча (Blocker)

Блокуюча помилка, що приводить застосування в неробочий стан, в результаті якого подальша робота з тестованою системою або її ключовими функціями стає неможлива. Вирішення проблеми необхідне для подальшого функціонування системи.

S2 Критична (Critical)

Критична помилка, неправильно працююча ключова бізнес логіка, дірка в системі безпеки, проблема, що привела до тимчасового падіння сервера або приводить в неробочий стан деяку частину системи, без можливості вирішення проблеми, використовуючи інші вхідні крапки. Вирішення проблеми необхідне для подальшої роботи з ключовими функціями тестованою системою.

S3 Значна (Major)

Значна помилка, частина основний бізнес логіки працює некоректно. Помилка не критична або є можливість для роботи з тестованою функцією, використовуючи інші вхідні крапки.

S4 Незначна (Minor)

Незначна помилка, що не порушує бізнес логіку тестованої частини застосування, очевидна проблема призначеного для користувача інтерфейсу.

S5 Тривіальна (Trivial)

Тривіальна помилка, що не стосується бізнес логіки застосування, погано відтворна проблема, малопомітна по засобах призначеного для користувача інтерфейсу, проблема сторонніх бібліотек або сервісів, проблема, що не робить ніякого впливу на загальну якість продукту.

Градація Пріоритету дефекту (Priority)

P1 Високий (High)

Помилка повинна бути виправлена щонайшвидше, оскільки її наявність є критичною для проекту.

P2 Середній (Medium)

Помилка повинна бути виправлена, її наявність не є критичною, але вимагає обов'язкового рішення.

P3 Низький (Low)

Помилка повинна бути виправлена, її наявність не є критичною, і не вимагає термінового рішення.

Порядок виправлення помилок по їх пріоритетах:

High -> Medium -> Low

Вимоги до кількості відкритих багів

Хочемо запропонувати вам наступний підхід до визначення вимог до кількості відкритих багів:

- Наявність відкритих дефектів P1, P2 і S1, S2, вважається неприйнятною для проекту. Всі подібні ситуації вимагають термінового рішення і йдуть під контроль до менеджерів проекту.
- Наявність строго обмеженої кількості відкритих помилок P3 і S3, S4, S5 не є критичним для проекту і допускається у видаваному застосуванні. Кількість же відкритих помилок залежить від розміру проекту і встановлених критеріїв якості.

Всі вимоги до відкритих помилок обмовляються і документуються на етапі ухвалення рішення про якість продукту, що розробляється. Як приклад документування подібних вимог - це пункт Критерии окончания тестирования у плані тестування.

Напісання баг репортів

Баг репорт - це технічний документ і у зв'язку з цим хочемо відзначити, що мова опису проблеми повинна бути технічною. Повинна використовуватися правильна термінологія при використанні назв елементів призначеного для користувача інтерфейсу (editbox, listbox, combobox, link, text area, button, menu, popup menu, title bar, system tray і так далі), дій користувача (click link, press the button, select menu item і так далі) і отриманих результатах (window is opened, error message is displayed, system crashed і так далі).

Вимоги до обов'язкових полів баг репорту

Відзначимо, що обов'язковими полями баг репорту є: **короткий опис** (*Bug Summary*), **серйозність** (*Severity*), **кроки до відтворення** (*Steps to reproduce*), **результат** (*Actual Result*), **очікуваний результат** (*Expected Result*). Нижче приведені вимоги і приклади по заповненню цих полів.

Короткий опис

Назва говорить само за себе. У одному пропозиція вам треба умістити сенс всього баг репорту, а саме: коротко і ясно, використовуючи правильну термінологію сказати що і де не працює. Наприклад:

1. Застосування зависає, при спробі збереження текстового файлу розміром більше 50Мб.
2. Дані на формі "Профайл" не зберігаються після натиснення кнопки "Зберегти".

На додаток пропонуємо вам вивчити Принцип "Где? Что? Когда?"описаний на сторінках блога "**QA Nest**":

"*У чому цей принцип полягає?*

Складіть пропозицію, в якій факти дефекту викладені в наступній послідовності:

- ***Де?**: У якому місці інтерфейсу користувача або архітектури програмного продукту знаходиться проблема. Причому, починайте пропозицію з іменника, а не приводу.*
- ***Що?**: Що відбувається або не відбувається згідно специфікації або вашому уявленню про нормальну роботу програмного продукту. При цьому указуйте на наявність або відсутність об'єкту проблеми, а не на його зміст (його указують в описі). Якщо зміст проблеми варіюється, всі відомі варіанти указуються в описі.*
- ***Коли?**: У який момент роботи програмного продукту, по настанню якої події або за яких умов проблема виявляється.*

Чому послідовність повинна бути саме такій?

У такому вигляді незнайомі дефекти зручніше сортувати по summary як показує практика (адже, швидше за все, саме серед дефектів інших інженерів проводитиметься пошук дублікатів). Якщо ви іншої думки - придумайте свою послідовність, але вона повинна стати єдиною для всіх без виключення членів проекту, інакше ви не доб'єтеся необхідного результату."

Серйозність

Якщо проблема знайдена в ключовій функціональності додатку і після її виникнення додаток стає повністю недоступним, і подальша робота з ним неможлива, то вона **блокуюча**. Зазвичай всі блокуючі проблеми знаходяться під час первинної перевірки нової версії продукту (**Build Verification Test Smoke Test**), оскільки їх наявність не дозволяє повноцінно проводити тестування. Якщо ж тестування може бути продовжене, то серйозність даного дефекту буде **критична**. На рахунок **значних, незначних і тривіальних** помилок питання достатньо прозоре і на наш погляд не вимагає зайвих пояснень.

Кроки до відтворення / Результат / Очікуваний результат

Дуже важливо чітко описати всі кроки, із згадуємо всіх даних (імені користувача, даних для заповнення форми), що вводяться, і проміжних результатів.

Наприклад:

Кроки до відтворення

1. Увійдіть до систем: Користувач Тестер1, пароль xxxXXX
--> Вхід в систему здійснений
2. Кликніть линк Профайл
--> Сторінка Профайл відкрилася
3. Введіть Нове ім'я користувача: Тестер2
4. Натисніть кнопку Зберегти

Результат

На екрані з'явилася помилка. Нове ім'я користувача не було збережене

Очікуваний результат

Сторінка профайл перевантажилася. Нове значення імені користувача збережене.

Основні помилки при написанні багів репортів

Недостатність наданих даних

Не завжди одна і таже проблема виявляється при всіх значеннях, що вводяться, і під будь-яким користувачем, що увійшов до системи, тому настійно рекомендується вносити всі необхідні дані до баг репорту

Визначення серйозності

Дуже часто відбувається або завищення, або заниження серйозності дефекту, що може привести до неправильної черговості при вирішенні проблеми.

Мова опису

Часто при описі проблеми використовуються неправильна термінологія або складні мовні звороти, які можуть ввести в оману людину, відповідальну за вирішення проблеми.

Відсутність очікуваного результату

У випадках, якщо ви не вказали, що ж повинно бути необхідною поведінкою системи, ви витрачаєте час розробника, на пошук даної інформації, тим самим уповільнюєте виправлення дефекту. Ви повинні вказати пункт у вимогах, написаний тест кейс або ж ваша особиста думка, якщо ця ситуація не була документована.

Заповнення полів баг репорту

У описаній нижче таблиці представлені основні поля баг репорту і роль працівника, відповідального за заповнення даного поля. Червоним кольором виділені обов'язкові для заповнення поля:

Поле	Поле
Короткий опис (Summary)	Автор баг репорту (звичайно це Тестувальник)
Проект (Project)	Автор баг репорту (звичайно це Тестувальник)
Компонент застосування (Component)	Автор баг репорту (звичайно це Тестувальник)
Номер версії (Version)	Автор баг репорту (звичайно це Тестувальник)
Серйозність (Severity)	Автор баг репорту (звичайно це Тестувальник), проте даний атрибут може бути змінений вищестоящим менеджером
Пріоритет (Priority)	Менеджер проекту або менеджер відповідальний за розробку компоненту, на який написаний баг репорт
Статус (Status)	Автор баг репорту (звичайно це Тестувальник), але багато систем баг трекінгу виставляють статус за умовчанням
Автор (Author)	Встановлюється за умовчанням, якщо немає, то указується ім'я автора баг репорту
Призначений на (Assigned To)	Менеджер проекту або менеджер відповідальний за розробку компоненту, на який написаний баг репорт
ОС / Сервіс Пак і так далі / Браузер + версія / ...	Автор баг репорту (звичайно це Тестувальник)
Кроки відтворення (Steps to Reproduce)	Автор баг репорту (звичайно це Тестувальник)
Фактичний Результат (Result)	Автор баг репорту (звичайно це Тестувальник)
Очікуваний результат (Expected Result)	Автор баг репорту (звичайно це Тестувальник)
Прикріплений файл (Attachment)	Автор баг репорту (звичайно це Тестувальник), а також будь-який член командної групи, що вважає, що прикріплені дані допоможуть у виправленні бага

Прочитавши короткий опис бага (Bug Summary), розробник повинен зрозуміти в чому полягає проблема, прочитавши детальний опис бага (Bug Description) - повинен знати рядок коду, який правити.

З цим можна погоджуватися або не погоджуватися, але сенс цього вислову в тому, що ви повинні робити все так, щоб до вас менше було питань по суті описаної в баг репорті проблеми. Оскільки кожен повернений вам баг репорт із статусом "Відхилений", "Не відтворюється", "Потрібна інформація" (Rejected, Can't Reproduce, More info) - це втрата часу, як вашого так і розробника. А час, як відомо - це гроші, які ми отримуємо, за те що робимо нашу роботу краще за всіх!

Упевнені, що якщо Ви в майбутньому скористаєтеся всіма запропонованими нами рекомендаціями, то якість Ваших баг репортів буде на високому рівні, і в процесі роботи до вас буде менше всього претензій, як від менеджерів, так від розробників.

- Тестове Покриття (Test Coverage) - це набір тестів для перевірки тестованої функції. Розрахунок тестового покриття проводиться по формулі: відношення кол-ва рядків коди, покритих тестами, до загального кол-ву рядків коди тестованої функції, помножене на 100%
- Деталізація Тест Кейсів (Test Case Detalization) - це рівень деталізації опису тестових кроків і необхідного результату, при якому забезпечується розумне співвідношення часу проходження до тестового покриття

Час Проходження Тест Кейса (Test Case Pass Time) - це час від початку проходження кроків тест кейса до отримання результату тіста.

ТЕМА 5. ТЕХНІКИ ТЕСТУВАННЯ

5.1. Техніка, що базується на інтуїції і досвіді інженера (Based on the software engineer's intuition and experience)

Спеціалізоване тестування (Ad hoc testing)

Можливо, найбільш застосована техніка. Тести грунтуються на досвіді, інтуїції і знаннях інженера, що розглядає проблему з погляду аналогій, що були раніше. Даний вид тестування може бути корисний для ідентифікації тих тестів, які не охоплюються більш формалізованою технікою.

Дослідницьке тестування (Exploratory testing)

Таке тестування визначається як одночасне навчання, проектування тесту і його виконання. Даний вид тестування заздалегідь не визначається в плані тестування і такі тести створюються, виконуються і модифікуються динамічно, в міру необхідності. Ефективність дослідницьких тестів безпосередньо залежить від знань інженера, що формуються на основі поведінки тестованого продукту в процесі проведення тестування, ступеня знайомства із застосуванням, платформою, типами можливих збоїв і дефектів, ризиками, що асоціюються з конкретним продуктом і тому подібне

5.2 Техніка, що базується на специфікації (Specification-based techniques)

Еквівалентне розділення <застосування> (Equivalence partitioning)

Дана область застосування розділяється на колекцію наборовши або еквівалентних класів, які вважаються еквівалентними з погляду даних зв'язків і характеристик <специфікації>. Репрезентативний набір тестів (іноді – тільки один тест) формується з тестів еквівалентних класів (або наборів класів).

Аналіз граничних значень (Boundary-value analysis)

Тести будуються з орієнтацією на використання тих величин, які визначають граничні характеристики тестованої системи. Розширенням цієї техніки є тести оцінки живучості (robustness testing) системи, що проводяться з величинами, що виходять за рамки специфікованих меж значень.

Таблиці ухвалення рішень (Decision table)

Такі таблиці представляють логічні зв'язки між умовами (можуть розглядатися як "входи") і діями (можуть розглядатися як "виходи"). Набір тестів будується послідовним розглядом всіх можливих кросс-связей в такій таблиці.

Тести на основі кінцевого автомата (Finite-state machine-based)

Будуються як комбінація тестів для всіх станів і переходів між станами, представлених у відповідній моделі (переходів і станів застосування).

Тестування на основі формальної специфікації (Testing from formal specification)

Для специфікації, визначених з використанням формальної мови, можливо автоматично створювати і тести для функціональних вимог. У ряді випадків можуть будується на основі моделі, частиною специфікації, що не використовує формальної мови опису, що є.

Випадкове тестування (Random testing)

На відміну від статистичного тестування (розглядатиметься в 5.1.5.1 "Operational profile"), самі тести генеруються випадковим чином за списком заданого набору специфікованих характеристик.

5.3 Техніка, орієнтована на код (Code-based techniques)

Тести, що базуються на блок-схемі (Control-flow-based criteria)

Набір тестів будується виходячи з покриття всіх умов і вирішень блок-схеми. Якоюсь мірою нагадує тести на основі кінцевого автомата. Відмінність – в джерелі набору тестів.

Максимальна віддача від тестів на основі блок-схеми виходить коли тести покривають різні шляхи блок-схеми – по-сути, сценарії потоків робіт (поведінки) тестованої системи. Адекватність таких тестів оцінюється як відсоток покриття всіх можливих шляхів блок-схеми.

Тести на основі потоків даних (Data-flow-based criteria)

В даних тестах відстежується повний життєвий цикл величин (змінних) – з моменту народження (визначення), на всьому протязі використання, аж до знищення (невизначеності). У реальній практиці використовуються нестроге тестування такого вигляду, орієнтоване, наприклад, тільки на перевірку завдання початкових значень всіх змінних або всіх входжень змінних в код, з погляду їх використання.

Посилальні моделі для тестування, орієнтованого на код (Reference models for code-based testing – flowgraph, call graph)

Є не стільки технікою тестування, скільки контролем структури програми, представленої у вигляді дерева викликів (наприклад, sequence-диаграммы, визначеної в нотації UML і побудованої на основі аналізу коди).

5.4 Тестування, орієнтоване на дефекти (Fault-based techniques)

Як це не дивно звучить на рівні назви такої техніки тестування, вони, дійсно, орієнтовані на помилки. Точніше – на специфічні категорії помилок.

Припущення помилок (Error guessing)

Направлені на виявлення найбільш вірогідних помилок, що передбачаються, наприклад, в результаті аналізу рисок.

Тестування мутацій (Mutation testing)

Мутація – невелика зміна тестованої програми, подія за рахунок приватних синтаксичних змін коди (зокрема, рефакторинга). Відповідні тести запускаються для оригінального і всіх варіантів тестованої програми, що “мутують”.

SWEBOK фокусується на можливості, за допомогою тестів, визначати відмінності між мутантами і початковим варіантом коди. Якщо така відмінність встановлена, мутанта “вбивають”, а тест вважається успішним. Зазвичай, даний підхід фокусується на синтаксичних помилках, що на практиці відстежуються сучасними середовищами розробки і, звичайно, компіляторами.

5.5 Техніка, що базується на умовах використання (Usage-based techniques)

Операційний профіль (Operational profile)

Базується на умовах використання системи.

Тестування для оцінки надійності системи повинне проводитися в такому тестовому оточенні, яке максимально наближене до реальних умов роботи системи. Результати таких тестів дозволяють оцінити поведінку системи в реальних умовах. Вхідні параметри тестів задаються на основі імовірнісного розподілу відповідних параметрів або їх наборів при експлуатації (вхідні дані можуть прогнозуватися виходячи з частоти можливих сценаріїв роботи користувачів).

Тестування, що базується на надійності інженерного процесу (**Software Reliability Engineered Testing**)

Базується на умовах розробки системи.

Відповідні тести (що позначаються також абревіатурою **SRET**) проектуються в контексті використовуваного процесу розробки і методик тестування.

5.6 Техніка, що базується на природі застосування (Techniques based on the nature of the application)

Описана техніка може застосовуватися до будь-яких типів програмних систем. В той же час, залежно від технологічної або архітектурної природи застосувань, можуть також застосовувати специфічну техніку, важливу саме для заданого типу застосування. Серед таких технік:

- Об'єктно-орієнтоване тестування
- Компонентно-орієнтоване тестування
- Web-орієнтоване тестування
- Тестування на відповідність протоколам
- Тестування систем реального часу

5.5. Вибір і комбінація різної техніки (Selecting and combining techniques)

Функціональне і структурне (Functional and structural)

Техніка тестування, що будуються на основі специфікацій або коди часто називають функціональними або структурними, відповідно. Обидва підходи не повинні протиставлятися, але доповнювати один одного.

Визначене або випадкове (Deterministic vs. random)

Зазвичай тести можна розподілити по даних групах на основі використовуваної політики вибору або визначення вхідних параметрів тестів.

Статистичне тестування. Операційний профіль. Функціональне і структурне тестування. Тестування об’єктно-орієнтованих програм. Тестування веб-застосувань. Тестування графічного інтерфейсу користувача. Тестування систем реального часу. Тестування критичних систем. Класифікація інструментів тестування. Вибір інструментів тестування. Методи аналізу показників функціональності. Методи вимірювання, що базуються на концепції функціонального розміру. Визначення розміру веб-застосувань. Стандартизація методів вимірювання розміру. Огляд основних методів оцінки затрат.

5.6. Види тестування програмного забезпечення

Всі **види тестування програмного забезпечення**, залежно від переслідуваних цілей, можна умовно розділити на наступні групи:

1. Функціональні
2. Нефункціональні
3. Пов'язані із змінами

Далі, ми постараємося детальніше розповісти про кожен окремий вид тестування, його призначенні і використанні при тестуванні програмного забезпечення.

Функціональні види тестування

Функціональні тести базуються на функціях і особливостях, а також взаємодії з іншими системами, і можуть бути представлені на всіх рівнях тестування: **компонентному або модульному (Component/Unit testing), інтеграційному (Integration testing), системному (System testing) і приймальному (Acceptance testing)**. Функціональні види тестування розглядають зовнішню поведінку системи. Далі перераховані одні з найпоширеніших видів функціональних тестів:

- Функціональне тестування (Functional testing)
- Тестування безпеки (Security and Access Control Testing)
- Тестування взаємодії (Interoperability Testing)

Нефункціональні види тестування

Нефункціональне тестування описує тести, необхідні для визначення характеристик програмного забезпечення, які можуть бути зміряні різними величинами. В цілому, це тестування того, "Як" система працює. Далі перераховані основні види нефункціональних тестів:

- Всі види тестування продуктивності:
  - тестування навантаження (Performance and Load Testing)
  - стресове тестування (Stress Testing)
  - тестування стабільності або надійності (Stability / Reliability Testing)
  - об'ємне тестування (Volume Testing)
- Тестування установки (Installation testing)
- Тестування зручності користування (Usability Testing)
- Тестування на відмову і відновлення (Failover and Recovery Testing)
- Конфігураційне тестування (Configuration Testing)

Пов'язані із змінами види тестування

Після проведення необхідних змін, таких як виправлення бага/дефекта, програмне забезпечення повинні бути пері тестовано для підтвердження того факту, що проблема була дійсно вирішена. Нижче перераховані види тестування, які необхідно проводити після установки програмного забезпечення, для підтвердження працездатності застосування або правильності здійсненого виправлення дефекту:

- Димове тестування (Smoke Testing)
- Регресійне тестування (Regression Testing)
- Тестування збірки (Build Verification Test)
- Санітарне тестування або перевірка узгодженості/справності (Sanity Testing)

Функціональні види тестування

Функціональні тести базуються на функціях і особливостях, а також взаємодії з іншими системами, і можуть бути представлені на всіх рівнях тестування:

Компонентне або модульне (Component/Unit testing)

Компонентне (модульне) тестування перевіряє функціональність і шукає дефекти в частинах застосування, які доступні і можуть бути протестовані по-отдельности (**модулі програм, об'єкти, класи, функції і так далі**). Звичайне компонентне (модульне) тестування проводиться викликаючи код, який необхідно перевірити і за підтримки середовищ розробки, таких як фреймворки (frameworks - каркаси) для модульного тестування або інструменти для відладки. Всі знайдені дефекти як правило виправляються в коді без формального їх опису в системі менеджменту багів (Bug Tracking System).

Один з найбільш ефективних підходів до компонентного (модульному) тестування - це підготовка автоматизованих тестів до початку основного кодування (розробки) програмного забезпечення. Це називається розробка від тестування (test-driven development) або підхід тестування спочатку (test first approach). При цьому підході створюються і інтегруються невеликі шматки коди, напроти яких запускаються тести, написані до початку кодування. Розробка ведеться до тих пір, поки всі тести не будуть успішними. Різниця між компонентним і модульним тестуванням

По-существу ці рівні тестування представляють одне і теж, різниця лише в тому, що в компонентному тестуванні як параметри функцій використовують реальні об'єкти і драйвери, а в модульному тестуванні - конкретні значення.

Інтеграційне тестування (Integration testing)

Інтеграційне тестування призначене для перевірки зв'язку між компонентами, а також взаємодії з різними частинами системи (операційною системою, устаткуванням або зв'язки між різними системами). Рівні інтеграційного тестування:

Компонентний інтеграційний рівень (Component Integration testing) Перевіряється взаємодія між компонентами системи після проведення компонентного тестування.

Системний інтеграційний рівень (System Integration Testing) Перевіряється взаємодія між різними системами після проведення системного тестування. Підходи до інтеграційного тестування:

Від низу до верху (Bottom Up Integration) Всі низькорівневі модулі, процедури або функції збираються воєдино і потім тестуються. Після чого збирається наступний рівень модулів для про ведення інтеграційного тестування. Даний підхід вважається корисним, якщо все або практично всі модулі, рівня, що розробляється, готові. Також даний підхід допомагає визначити за наслідками тестування рівень готовності застосування (див. також Integration testing - Bottom Up)

Зверху вниз (Top Down Integration) Спочатку тестуються всі високорівневі модулі, і поступово один за іншим додаються низькорівневі. Всі модулі нижчого рівня сиулируются заглушками з аналогічною функціональністю, навіщо у міру готовності вони замінюються реальними активними компонентами. Таким чином ми проводимо тестування зверху вниз. (див. також Top Down Integration)

Великий вибух ("Big Bang" Integration) Все або практично всі розроблені модулі збираються разом у вигляді закінченої системи або її основної частини, і потім проводиться інтеграційне тестування. Такий підхід дуже хороший для збереження часу. Проте якщо тест кейси і їх результати записані не вірно, то сам процес інтеграції сильно ускладниться, що стане перешкодою для команди тестування досягши основної мети інтеграційного тестування (див. також Integration testing - Big Bang)

Системне тестування (System testing)

Основним завданням системного тестування є **перевірка як функціональних, так і не функціональних вимог в системі в цілому.** При цьому виявляються дефекти, такі як невірне використання ресурсів системи, непередбачені комбінації даних призначеного для користувача рівня, несумісність з оточенням, непередбачені сценарії використання, відсутня або невірна функціональність, незручність використання і так далі Для мінімізації рисок, пов'язаних з особливостями поведінки в системи в тому або іншому середовищі, **під час тестування рекомендується використовувати оточення максимально наближене до того, на яке буде встановлений продукт після видачі.**

Можна виділити два підходи до системного тестування:

- **на базі вимог** (requirements based) Для кожної вимоги пишуться тестові випадки (test cases), перевіряючі виконання даної вимоги.
- **на базі випадків використання** (use case based)

На основі уявлення про способи використання продукту створюються випадки використання системи (Use Cases). По конкретному випадку використання можна визначити один або больее сценаріїв. На перевірку кожного сценарію пишуться тест кейси (test cases), які повинні бути протестовані.

Приймальне тестування(Acceptance testing).

Формальний процес тестування, який перевіряє відповідність системи вимогам і проводиться з метою:

- визначення чи задовольняє систему приймальних критеріям;
- винесення ухвали замовником або іншою уповноваженою особою приймається застосування чи ні.

Приймальне тестування виконується на підставі набору типових тестових випадків і сценаріїв, розроблених на підставі вимог до даного застосування.

Рішення про проведення приймального тестування ухвалюється, коли:

- продукт досяг необхідного рівня якості;
- замовник ознайомлений з Планом Приймальних Робіт (Product Acceptance Plan) або іншим документом, де описаний набір дій, пов'язаних з проведенням приймального тестування, дата проведення, відповідальні і так далі

Фаза приймального тестування триває до тих пір, поки замовник не виносить ухвалу про відправлення застосування на доопрацювання або видачу застосування. Шаблон плану приймально-здавальних випробувань від RUP можна викачати, кликнувши по посиланню: RUP Product Acceptance Plan

Функціональні види тестування розглядають зовнішню поведінку системи. Далі перераховані одні з найпоширеніших видів функціональних тестів:

Функціональне тестування (Functional testing)

Тестування безпеки (Security and Access Control Testing)

Стратегія тестування, використовувана для перевірки безопаности системи, а також для аналізу рисок, пов'язаних із забезпеченням цілісного підходу до захисту застосування, атак хакерів, вірусів, несанкціонованого доступу до конфіденційних даних. Тестування безпеки може виконуватися як автоматизований так і в ручну, включаючи перевірку як позитивних, так і негативних тестових випадків. Основививається на трьох основних принципах - **це конфіденційність, цілісність і доступність** (confidentiality, integrity, availability)

Конфіденційність - це захования певних ресурсів або інформації. Під конфіденційністю можна розуміти обмеження доступу до ресурсу деякої категорії користувачів, або іншими словами, за яких умов користувач авторизований дістати доступ до даного ресурсу.

Існує два основні критерії при визначенні поняття цілісності:

Довіра. Очікується, що ресурс буде змінений тільки відповідним способом певною групою користувачів.

Пошкодження і відновлення. У разі коли дані ушкоджуються або неправильно міняються авторизованим або не авторизованим користувачем, ви повинні визначити на скільки важливою є процедура відновлення даних.

Доступність є вимогами про те, що ресурси повинні бути доступні авторизованому користувачеві, внутрішньому об'єкту або пристрою. Як правило, ніж критичніший ресурс тем выше уровень доступности должен быть.

Тестування взаємодії (Interoperability Testing)

З розвитком мережевих технологій і інтернету взаємодія різних систем, сервісів і застосувань один з одним придбало значну актуальність, оскільки будь-які пов'язані з цим проблеми можуть привести до падіння авторитету компанії, що як наслідок спричинить фінансові втрати. Тому до тестування взаємодії варто підходити зі всією серйозністю.

Тестування взаємодії (Interoperability Testing) – це функціональне тестування, перевіряюче здатність застосування взаємодіяти з одним і більш компонентами або системами і тестування сумісності (compatibility testing), що включає, і інтеграційне тестування (integration testing).

Програмне забезпечення з хорошими характеристиками взаимодествия може бути легко інтегроване з іншими системами, не вимагаючи яких-небудь серйозних модифікацій. В цьому випадку, кількість змін і час, потрібний на їх виконання, можуть бути використані для вимірювання можливості взаємодії.

Нефункціональне тестування

Нефункціональне тестування описує тести, необхідні для визначення характеристик програмного забезпечення, які можуть бути зміряні різними величинами. В цілому, це тестування того, "Як" система працює. Далі перераховані основні види нефукнциональных тестів:

Всі види тестування продуктивності:

- тестування навантаження (Performance and Load Testing)
- стресове тестування (Stress Testing)
- тестування стабільності або надійності (Stability / Reliability Testing)
- об'ємне тестування (Volume Testing)

Тестування продуктивності (Performance testing)

Завданням тестування продуктивності є визначення масштабованості застосування під навантаженням, при цьому відбувається:

- вимірювання часу виконання вибраних операцій при певних інтенсивностях виконання цих операцій
- визначення кількості користувачів, що одночасно працюють із застосуванням
- визначення меж прийнятної продуктивності при збільшенні навантаження (при збільшенні інтенсивності виконання цих операцій)
- дослідження продуктивності на високих, граничних, стресових навантаженнях

Стресове тестування

Стресове тестування дозволяє перевірити наскільки застосування і система в цілому працездатні в умовах стресу і також оцінити здатність системи до регенерації, тобто до повернення до нормального стану після припинення дії стресу. Стресом в даному контексті може бути підвищення інтенсивності виконання операцій до дуже високих значень або аварійна зміна конфігурації сервера. Також одному із завдань при стресовому тестуванні може бути оцінка деградації продуктивності, таким чином цілі стресового тестування можуть перетинатися з цілями тестування продуктивності.

Тестування об'єму

Завданням **об'ємного тестування** є отримання оцінки продуктивності при збільшенні об'ємів даних в базі даних застосування, при цьому відбувається:

вимірювання часу виконання вибраних операцій при визначених інтенсивності виконання цих операцій

може проводитися визначення кількості користувачів, що одночасно працюють із застосуванням

Тестування стабільності або надійності (Stability / Reliability Testing) Завданням тестування стабільності (надійності) є перевірка працездатності застосування при тривалому (багатогодинному) тестуванні з середнім рівнем навантаження. Часи виконання операцій можуть грати в даному виді тестування другорядну роль. При цьому на перше місце виходить відсутність витоків пам'яті, перезапусків серверів під навантаженням і інші аспекти впливають саме на стабільність роботи.

Тестування реакції системи на зміну навантаження

Load vs Performance Testing — В англомовній термінології ви можете так само знайти ще один вид тестування - Load Testing - **тестування реакції системи на зміну навантаження** (у межі допустимого). Нам здалося, що Load і Performance переслідують все ж таки одну і ту ж мету: перевірка продуктивності (часів відгуку) на різних навантаженнях. Власне тому ми і не почали розділяти їх. В той же час хто те може розділити. Головне все-таки розуміти цілі того або іншого виду тестування і постаратися їх досягти.

Теорія і практика — Докладну інформацію про те, що таке тестування навантаження і тестування продуктивності програм, а так само інформацію про методику проведення, ви можете дізнатися в розділі Автоматизація тестування навантаження

Тестування установки (Installation testing)

Тестування установки направлено на перевірку успішної інсталяції і настройки, а також оновлення або видалення програмного забезпечення. Зараз найбільш поширена установка ПЗ за допомогою **инсталляторов** (спеціальних програм, **які самі по собі** так само **вимагають належного тестування**).

У реальних умовах инсталляторов може не бути. В цьому випадку доведеться самостійно виконувати установку програмного забезпечення, використовуючи документацію у вигляді інструкцій або readme файлів, що крок за кроком описують всі необхідні дії і перевірки.

У розподілених системах, де застосування розгортається на вже працюючому оточенні, простого набору інструкцій може бути мало. Для цього, часто, пишеться план установки (Deployment Plan), що включає не тільки кроки по інсталяції застосування, але і кроки відкоту (roll-back) до попередньої версії, у разі невдачі. Сам по собі **план установки також повинен пройти процедуру тестування** для уникнення проблем при видачі в реальну експлуатацію. Особливо це актуально, якщо установка виконується на системи, де кожна хвилина простою - це втрата репутації і великої кількості засобів, наприклад: банки, фінансові компанії або навіть баннерные мережі. Тому тестування установки можна назвати одному з найважливіших завдань по забезпеченню якості програмного забезпечення.

Саме такий комплексний підхід з написанням планів, покроковою перевіркою установки і відкоту інсталяції, повноправно можна назвати **тестуванням установки** або Installation Testing.

Тестування зручності користування (Usability Testing)

Іноді ми стикаємося з незрозумілими, нелогічними застосуваннями, великою кількістю функцій способи використання яких часто не очевидні. Після такої роботи рідко виникає

бажання використовувати застосування знову, і ми шукаємо зручніші аналоги. Для того, щоб застосування було популярним, йому мало бути функціональним – воно повинне бути ще і зручним. Якщо задуматися, інтуїтивно зрозумілі застосування економлять нерви користувачам і витрати працедавця на навчання. А значить вони більш конкурентноспроможні. Тому тестування зручності використання, про яке піде мова далі є невід'ємною частиною тестування будь-яких масових продуктів.

Тестування зручності користування - це метод тестування, направлений на встановлення ступеня зручності використання, навчання, зрозумілості і привабливості для користувачів продукту, що розробляється, в контексті заданих умов. [ISO 9126]

Тестування зручності користування дає оцінку рівня зручності використання застосування за наступними пунктами:

- **продуктивність, ефективність** (efficiency) - скільки часу і кроків знадобиться користувачеві для завершення основних завдань застосування, наприклад, розміщення новини, реєстрації, покупка і т.д. (менше - краще);
- **правильність** (accuracy) - скільки помилок зробив користувач під час роботи із застосуванням (менше - краще);
- **активізація в пам'яті** (recall) – як багато користувач пам'ятає про роботу застосування після припинення роботи з ним на тривалий період часу (повторне виконання операцій після перерви повинне проходити швидше чим у нового користувача);
- **емоційна реакція** (emotional response) – як користувач себе відчуває після завершення завдання - розгублений, випробував стрес. Чи порекомендує користувач систему своїм друзям? (позитивна реакція - краще)

Перевірка зручності використання може проводитися як по відношенню до готового продукту, за допомогою тестування чорного ящика (black box testing), так і до інтерфейсів застосування (API), які використовуються при розробці, - тестування білого ящика (white box testing). В цьому випадку перевіряється зручність використання внутрішніх об'єктів, класів, методів і змінних, а також розглядається зручність зміни, розширення системи і інтеграції її з іншими модулями або системами. Використання зручних інтерфейсів (API) може поліпшити якість, збільшити швидкість написання і підтримки коди, що розробляється, і як наслідок поліпшити якість продукту в цілому.

Звідси стає очевидно, що тестування зручності користування може проводитися на різних рівнях розробки програмного забезпечення: модульному інтеграційному системному. При цьому воно цілком і повністю буде залежить від того, хто використовуватиме застосування на виділеному конкретному рівні - розробник, бізнес користувач системи і так далі

Поради з поліпшення зручності користування

Для дизайну зручних застосувань корисно слідувати принципам «пока-йока» або fail-safe. У нас це більш відомо як «захист від дурня». Простий приклад, якщо поле вимагає цифрове значення, логічно обмежити користувачеві діапазон введення тільки цифрами – буде менше випадкових помилок.

Для покращення існуючих застосувань можна використовувати цикл Деммінга Plan-Do-Check-Act, збираючи відгуки про роботу і дизайн застосування у існуючих користувачів, і, відповідно до їх зауважень, плануючи і проводячи поліпшення.

Помилки при тестуванні зручності користування:

Тестування призначеного для користувача інтерфейсу = Тестування зручності користування

Тестування зручності користування не має нічого спільного з тестуванням функціональності призначеного для користувача інтерфейсу, воно лише проводиться на призначеному для користувача інтерфейсі так само як і на багатьох інших можливих компонентах продукту. При цьому тип тестування і тесткейсы будуть зовсім інші, оскільки мова може йти про зручність використання не візуальних компонентів (якщо такі є) або процес адміністрування, наприклад, розподіленого клієнт-серверного продукту і так далі

Тестування зручності користування можна провести без участі експерта Не завжди людина не розбирається в наочній області здатний провести його самостійно. Уявіть, що тестувальникові потрібно протестувати зручність користування стратегічного бомбардувальника. Йому доведеться перевірити основні функції: зручність ведення бою, навігація, пілотування, обслуговування, наземного транспортування і так далі

Тестування на відмову і відновлення (Failover and Recovery Testing)

Конфігураційне тестування (Configuration Testing)

Після проведення необхідних змін, таких як виправлення бага/дефекта, програмне забезпечення повинні бути пері тестовано для підтвердження того факту, що проблема була дійсно вирішена. Нижче перераховані види тестування, які необхідно проводити після установки програмного забезпечення, для підтвердження працездатності застосування або правильності здійсненого виправлення дефекту:

Димове тестування (Smoke Testing)

Поняття димове тестування пішло з інженерного середовища. При введенні в експлуатацію нового "заліза" вважалося, що тестування пройшло вдало, якщо з установки не пішов дим. У області ж тестування програмного забезпечення, воно направлене на поверхневу перевірку всіх модулів застосування на предмет працездатності і наявність швидко находимых критичних і блокуючих дефектів. За наслідками того, що димового третирує робиться вивід про те, приймається чи ні встановлена версія програмного забезпечення в тестування, експлуатацію або на постачання замовникові. Для полегшення роботи, економії часу і людських ресурсів рекомендується автоматизувати димові тести.

Регресійне тестування (Regression Testing)

Вид тестування направлений на перевірку змін, зроблених в застосуванні або навколишньому середовищі (лагодження дефекту, злиття коди, міграція на іншу операційну систему, базу даних, веб-сервер сервер або сервер застосування), для підтвердження того факту, що функціональність, що існує раніше, працює як і раніше. Регресійними можуть бути тести як функціональні, так і не функціональні.

Як правило, **для регресійного тестування використовуються тест кейси, написані на ранніх стадіях розробки і тестування.** Це дає гарантію того, що зміни в новій версії застосування не пошкодили вже існуючу функціональність. Рекомендується робити автоматизацію регресійних тестів, для прискорення подальшого процесу тестування і виявлення дефектів на ранніх стадіях розробки програмного забезпечення.

Термін "Регресійне тестування", залежно від контексту використання може мати різний сенс. Сем Канер, наприклад, описав 3 основних типу регресійного тестування:

Регресія багів (Bug regression) - спроба довести, що виправлена помилка насправді не виправлена

Регресія старих багів (Old bugs regression) - спроба довести, що недавню зміну коди або даних зламало виправлення старих помилок, тобто старі баги стали знову відтворюватися.

Регресія побічного ефекту (Side effect regression) - спроба довести, що недавня зміна коди або даних зламала інші частини застосування, що розроблялося

Тестування збору (Build Verification Test)

Тестування направлене на визначення відповідності, випущеної версії, критеріям якості для початку тестування. По своїх цілях є аналогом Димового Тестування направленого на приймання нової версії в подальше тестування або експлуатацію. Углиб воно може проникати далі, залежно від вимог до якості випущеної версії.

Санітарне тестування або перевірка узгодженості/справності (Sanity Testing)

Вузьконаправлене тестування достатнє для доказу того, що конкретна функція працює узгоджено заявленим в специфікації вимогам. Є підмножиною регресійного тестування. Використовується для визначення працездатності певної частини застосування після змін вироблених в ній або навколишньому середовищі. Зазвичай виконується уручну. Відмінність санітарного тестування від димового (Sanity vs Smoke testing) В деяких джерелах помилково вважають, що санітарне і димове тестування - це одне і теж. Проте ці види тестування мають "вектор руху", напрямлений в різні боки. У відмінності від Smoke testing

Sanity testing направлене на функції, що углиб перевіряються, тоді як димове направлене вшир, для покриття тестами як можна більшого функціоналу в найкоротші терміни.

Контрольні запитання

ТЕМА 6. ТЕСТУВАННЯ ПРИЗНАЧЕНОГО ДЛЯ КОРИСТУВАЧА ІНТЕРФЕЙСУ:

Лекція присвячена тестуванню призначеного для користувача інтерфейсу. Визначаються завдання і цілі даного виду тестування, визначаються методи функціонального тестування призначеного для користувача інтерфейсу, вводяться основні поняття тестування зручності використання (usability) інтерфейсів. Мета даної лекції: дати уявлення про процес тестування призначеного для користувача інтерфейсу, його завдання, цілі і основні методи.

6.1. Завдання і цілі тестування призначеного для користувача інтерфейсу

Частина програмної системи, що забезпечує роботу інтерфейсу з користувачем, - один з найбільш нетривіальних об'єктів для верифікації. Нетривіальність полягає в двоякому сприйнятті терміну "призначений для користувача інтерфейс".

З одного боку, призначений для користувача інтерфейс - частина програмної системи. Відповідно, на призначений для користувача інтерфейс пишуться функціональні і низькорівневі вимоги, по яких потім складаються тест-требования і тест-планы. При цьому, як правило, вимоги визначають реакцію системи на кожне введення користувача (за допомогою клавіатури, миші або іншого пристрою введення) і вид інформаційних повідомлень системи, що виводяться на екран, що друкує пристрій або інший пристрій виводу. При верифікації таких вимог мова йде про перевірці функціональної повноти призначеного для користувача інтерфейсу - наскільки реалізовані функції відповідають вимогам, чи коректно виводиться інформація на екран.

З іншого боку, призначений для користувача інтерфейс - "обличчя" системи, і від його продуманості залежить ефективність роботи користувача з системою. Чинники, що впливають на ефективність роботи, слабо піддаються формалізації у вигляді конкретних вимог до окремих елементів, проте повинні бути враховані у вигляді загальних рекомендацій і принципів побудови призначеного для користувача інтерфейсу програмної системи. Перевірка інтерфейсу на ефективність людино-машинної взаємодії отримала назву перевірки зручності використання (usability verification ; у російськомовній літературі як переклад терміну usability часто використовують слово "практичність").

У даній лекції будуть розглянуті загальні питання як функціонального тестування призначених для користувача інтерфейсів, так і тестування зручності використання.

6.2. Функціональне тестування призначених для користувача інтерфейсів

Функціональне тестування призначеного для користувача інтерфейсу складається з п'яти фаз:

- аналіз вимог до призначеного для користувача інтерфейсу;
- розробка тест-вимог і тест-планів для перевірки призначеного для користувача інтерфейсу;
- виконання тестових прикладів і збір інформації про виконання тестів;
- визначення повноти покриття призначеного для користувача інтерфейсу вимогами;
- складання звітів про проблеми у разі неспівпадання поведінки системи і вимог або у разі відсутності вимог на окремі інтерфейсні елементи.

Всі ці фази такі самі, як і у разі тестування будь-якого іншого компоненту програмної системи. Відмінності полягають в трактуванні деяких термінів в застосуванні до призначеного для користувача інтерфейсу і в особливостях автоматизованого збору інформації на кожній фазі.

Так, тест-планы для перевірки призначеного для користувача інтерфейсу, як правило, є сценарії, що описують дії користувача при роботі з системою. Сценарії можуть бути записані або на природній мові, або на формальній мові якої-небудь системи автоматизації призначеного для користувача інтерфейсу. Виконання тестів при цьому проводиться або оператором в ручному режимі, або системою, яка емулює поведінку оператора.

При зборі інформації про виконання тестових прикладів зазвичай застосовуються технології аналізу форм, що виводяться на екран, і їх елементів (у разі графічного інтерфейсу) або тексту (у разі текстового), що виводиться на екран, а не перевірка значень тих або інших змінних, що встановлюються програмною системою.

Під повнотою покриття призначеного для користувача інтерфейсу розуміється те, що в результаті виконання всіх тестових прикладів кожен інтерфейсний елемент був використаний хоч би один раз у всіх доступних режимах.

Звіти про проблеми в призначеному для користувача інтерфейсі можуть включати як описи невідповідностей вимог і реальної поведінки системи, так і описи проблем у вимогах до призначеного для користувача інтерфейсу. Основне джерело проблем в цих вимогах - їх тестонепригодность, викликана розпливчатістю формулювань і неконкретністю.

6.2.1. Перевірка вимог до призначеного для користувача інтерфейсу

6.2.1.1. Типи вимог до призначеного для користувача інтерфейсу

Вимоги до призначеного для користувача інтерфейсу можуть бути розбиті на дві групи:

- вимоги до зовнішнього вигляду призначеного для користувача інтерфейсу і формам взаємодії з користувачем;
- вимоги по доступу до внутрішньої функціональності системи за допомогою призначеного для користувача інтерфейсу.

Іншими словами, перша група вимог описує взаємодію підсистеми інтерфейсу з користувачем, а друга - з внутрішньою логікою системи.

До першої групи можна віднести наступні типи вимог.

- **Вимоги до розміщення елементів управління на екранних формах**

Дані вимоги можуть визначати загальні принципи розміщення елементів призначеного для користувача інтерфейсу або вимоги до розміщення конкретних елементів. Наприклад, загальні вимоги по розміщенню елементів на графічній екранній формі можуть виглядати таким чином:

Кожне вікно застосування повинне бути розбите на три частини: рядок меню, робоча область і статусний рядок. Рядок меню повинен бути горизонтальним і притиснутим до верхньої частини вікна, статусний рядок повинен бути горизонтальним і притиснутим до нижньої частини вікна, робоча область повинна знаходитися між рядком меню і статусним рядком і займати всю площу вікна, що залишилася.

При тестуванні даної вимоги досить визначити, що в кожному вікні системи дійсно присутньо три частини, які розташовані і притиснуті згідно вимогам навіть при зміні розмірів вікна, його згортанні/розгортанні, переміщенні по екрану, при перекритті його іншими вікнами.

Прикладом вимог по розміщенню конкретного елементу може служити наступне:

Кнопка "Почати передачу" повинна знаходитися безпосередньо під рядком меню в лівій частині робочої зони вікна.

При тестуванні такої вимоги також необхідно визначити, чи зберігається розташування елементу при зміні розміру вікна, а також при використанні елементу (в даному випадку - при натисненні).

- **Вимоги до змісту і оформлення повідомлень, що виводяться**

Вимоги до змісту і оформлення повідомлень, що виводяться, визначають текст повідомлень, що виводяться системою, його шрифтове і колірне оформлення. Також часто в таких вимогах визначається, в яких випадках виводиться те або інше повідомлення.

Так, наприклад, для тестування вимоги

Повідомлення "Неможливо відкрити файл" повинно виводитися в статусний рядок притиснутим до лівого краю, червоним кольором, напівжирним шрифтом у разі недоступності файлу, що відкривається, по читанню.

необхідно перевірити, що при виникненні вказаної ситуації повідомлення дійсно виводиться згідно вимогам.

Проте у разі тестування вимоги вигляду

Повідомлення про помилки повинні виводитися в статусний рядок притиснутими до лівого краю червоним кольором напівжирним шрифтом.

необхідно перевіряти формати всіх можливих повідомлень про помилки програми у всіх можливих помилкових ситуаціях. Таким чином, очевидно, що при тестуванні призначеного для користувача інтерфейсу не завжди можна однозначно визначити кількість тестових прикладів, які знадобляться для тестування вимоги. Ця проблема викликана тим, що вимоги до призначеного для користувача інтерфейсу часто здаються дуже очевидними для їх точного формулювання. Ця неконкретність вимог і викликає велику кількість тестів для кожної вимоги.

- **Вимоги до форматів введення**

Дана група вимог визначає, в якому вигляді інформація поступає від користувача в систему. При цьому окрім власне вимог, що визначають коректний формат, до цієї групи відносяться вимоги, що визначають реакцію системи на некоректне введення. Для перевірки таких вимог необхідно перевіряти як коректне введення, так і некоректний. Бажано при цьому розбивати різні варіанти введення на класи еквівалентності (як мінімум на два - коректні і некоректні).

До другої групи відносяться наступні типи вимог.

- **Вимоги до реакції системи на введення користувача**

Даний тип вимог визначає зв'язок внутрішньої логіки системи і інтерфейсних елементів. Наприклад

При натисненні кнопки "Скидання" значення таймера синхронізації передачі повинне скидатися в 0.

Для перевірки такої вимоги в тестовому прикладі повинне бути сымитировано натиснення на кнопку "Скидання", після чого повинна проводитися перевірка значення таймера. Проте деякі вимоги визначають як реакцію системи не те, як міняється її внутрішній стан, а реакцію призначеного для користувача інтерфейсу. Наприклад, у вимозі

При натисненні кнопки "Відкладене скидання" повинне виводитися вікно "Введення значення часу для відкладеного скидання".

як реакція на використання одного інтерфейсного елементу визначається поява іншого інтерфейсного елементу. Такі вимоги перевіряються за допомогою імітації введення користувача і аналізу інтерфейсних елементів, що з'являються.

- **Вимоги до часу відгуку на команди користувача**

Як окремий тип вимог можна виділити вимоги до часу відгуку системи на різні призначені для користувача операції. Це пов'язано з тим, що підсвідомо користувач сприймає операції тривалістю більше 1 секунди як тривалі. Якщо у цей момент система не повідомляє користувача про те, що вона виконує яку-небудь операцію, користувач почне вважати, що система зависла або працює в невірному режимі. У зв'язку з цим або кожен граничний час відгуку повинен бути вказане у вимогах і призначеній для користувача документації, або під час тривалих операцій повинні виводитися інформаційні повідомлення (наприклад, індикатор прогресу). Значення граничного часу і рівномірність збільшення значень індикатора прогресу повинні перевірятися відповідними тестами.

6.2.1.2. Тестопрігодность вимог до призначеного для користувача інтерфейсу

Деякі вимоги до призначеного для користувача інтерфейсу можуть опинитися тестонепригодными, або їх тестування буде значно утруднено. До таких вимог насамперед відносяться вимоги, що описують суб'єктивні характеристики інтерфейсу, які не можуть бути точно визначені або зміряні при виконанні тестових прикладів. При аналізі вимог до призначеного для користувача інтерфейсу необхідно чітко представляти, який елемент інтерфейсу і яким чином перевірятиметься, яка його характеристика вимірюватиметься в ході тестування.

Прикладом тестонепригодного вимоги може служити класична вимога

Призначений для користувача інтерфейс повинен бути інтуїтивно зрозумілим.

Без визначення чітких критеріїв інтуїтивної зрозумілості перевірка такої вимоги неможлива. При цьому необхідно розуміти, що критерій в даному випадку може бути двох видів: детермінованим або імовірнісним. Прикладом детермінованого критерію може бути доповнення до вимоги вигляду

Під інтуїтивною зрозумілістю інтерфейсу розуміється доступність будь-якої функції системи при допомозі не більше ніж 5 клацань миші по інтерфейсних елементах.

Вимога з таким уточненням піддається як ручному, так і автоматизованому тестуванню, більш того, результат такого тестування не залежатиме від суб'єктивної думки тестувальника (поняття про інтуїтивну зрозумілість у всіх разные).

Прикладом імовірнісного критерію може служити наступне доповнення:

Під інтуїтивною зрозумілістю інтерфейсу розуміється, що користувач звертається до керівництва користувача не частіше, ніж раз в п'ять хвилин на етапі навчання і не частіше, ніж раз о 2 годині на етапі активного використання системи. Значень повинні бути набуті на репрезентативній вибірці користувачів не менше 1000 чоловік.

Перевірка вимоги з таким доповненням не є завданням класичної верифікації і відноситься вже швидше до перевірки зручності застосування призначеного для користувача інтерфейсу. Проте тут також вводиться чіткий критерій, при використанні якого результати тестування можуть бути відтворені.

6.2.2. Повнота покриття призначеного для користувача інтерфейсу

При визначенні поняття покриття призначеного для користувача інтерфейсу можна ввести наступні його рівні:

- **функціональне покриття** - покриття вимог до призначеного для користувача інтерфейсу;
- **структурне покриття** - для забезпечення повного структурного покриття кожен інтерфейсний елемент повинен бути використаний в тестових прикладах хоч би один раз;
- **структурне покриття з урахуванням стану елементів інтерфейсу** - для забезпечення цього рівня покриття необхідно не тільки використовувати кожен елемент інтерфейсу, але і привести його у всі можливі стани (наприклад, для чек-боксов - отмечен/не відмічений, для полів введення - пустое/заполненное не целиком/заполненное повністю і тому подібне)
- **структурне покриття з урахуванням стану елементів інтерфейсу і внутрішнього стану системи** - поведінка деяких інтерфейсних елементів може змінюватися залежно від внутрішнього стану системи. Кожне така помітна поведінка інтерфейсного елементу повинна бути перевірене. Наприклад, система може мати два режими роботи - нормальний і для початкуючого користувача, в якому натиснення кожного елементу супроводжується появою спливаючої підказки. В цьому випадку потрібно перевірити обидва режими і при цьому перевірити, що підказки з'являються тільки в режимі для початківців.

При визначенні ступеня покриття необхідно враховувати, що реакція на деякі інтерфейсні елементи визначається не програмною системою, а на рівні операційної системи або середовища виконання. Так, наприклад, реакція на використання багатьох інтерфейсних елементів стандартного діалогового вікна відкриття файлу визначається операційною системою і може не тестуватися.

Якщо рівень покриття інтерфейсних елементів тестами недостатній, це є сигналом або до уточнення вимог до призначеного для користувача інтерфейсу, або до зниження ступеня подробиці тестування.

6.2.3. Методи проведення тестування призначеного для користувача інтерфейсу, повторюваність тестування призначеного для користувача інтерфейсу

Функціональне тестування призначеного для користувача інтерфейсу може проводитися різними методами - як уручну при безпосередній участі оператора, так і за

допомогою різного інструментарію, що автоматизує виконання тестових прикладів. Розглянемо ці методи детальніше.

6.2.3.1. Ручне тестування

Ручне тестування призначеного для користувача інтерфейсу проводиться тестувальником-оператором, який керується в своїй роботі описом тестових прикладів у вигляді набору сценаріїв. Кожен сценарій включає перерахування послідовності дій, які повинні виконати оператор, і опис важливих для аналізу результатів тестування у відповідь реакцій системи, відбиваних в призначеному для користувача інтерфейсі. Типова форма запису сценарію для проведення ручного тестування - таблиця, в якій в одній колонці описані дії (кроки сценарію), в іншій - очікувана реакція системи, а третя призначена для запису того, чи співпала очікувана реакція системи з реальною і перерахування неспівпадань (Таблица 6.1).

Таблиця 6.1. Приклад сценарію для ручного тестування призначеного для користувача інтерфейсу

№ п/п	Дія	Реакція системи	Результат
1	Клацніть на піктограмі System і виберіть пункт меню'System Management Applet'. Вікно має назву 'System Management Application	З'явиться вікно введення логіна і пароля	Вірно
2	Введіть у вікно введення, що з'явилося, ім'я користувача 'guest1'и пароль 'guest'. Потім натисніть кнопку 'Login'.	З'явиться вікно 'System Management Applet'. У верхньому правому кутку повинне бути виведене ім'я користувача guest1, що увійшов Всі опції у вікні повинні бути відключені (виведені сірим кольором)	Невірно
3	Завершите сеанс роботи з аплетом клацанням по пиктограмме'Logout'	Вікно 'System Management Applet' повинне бути закрите	Вірно

Ручне тестування призначеного для користувача інтерфейсу зручне тим, що контроль коректності інтерфейсу проводиться людиною, тобто основним "споживачем" даної частини програмної системи. До того ж при чисто косметичних змінах в інтерфейсах системи, не відбитих у вимогах (наприклад, при переміщенні кнопок управління на 10 пікселів вліво), аналіз успішності проходження тесту виконуватиметься не по формальних ознаках, а згідно людському сприйняттю.

При цьому ручне тестування має і істотний недолік - для його проведення потрібні значні людські і тимчасові ресурси. Особливо сильно цей недолік виявляється при проведенні регресійного тестування і взагалі будь-якого повторного тестування - на кожній ітерації повторного тестування призначеного для користувача інтерфейсу потрібна участь тестувальника-оператора. У зв'язку з цим в останнє десятиліття набули поширення засобу автоматизації тестування призначеного для користувача інтерфейсу, що знижують навантаження на тестувальника-оператора.

6.2.3.2. Сценарії на формальних мовах

Природний спосіб автоматизації тестування призначеного для користувача інтерфейсу - використання програмних інструментів, що емулюють поведінку тестувальника-оператора при ручному тестуванні призначеного для користувача інтерфейсу.

Такі інструменти використовують як вхідну інформацію сценарії тестових прикладів, записані на деякій формальній мові, оператори якої відповідають діям користувача - введенню команд, переміщенню курсора, активізації пунктів меню і інших інтерфейсних елементів.

При виконанні автоматизованого тесту інструмент тестування імітує дії користувача, описані в сценарії, і аналізує інтерфейсну реакцію системи. Для визначення очікуваного стану призначеного для користувача інтерфейсу тут можуть застосовуватися різні методи - або аналіз знімків екрану і порівняння їх з еталонними, або доступ до даним інтерфейсних елементів засобами операційної системи (наприклад, доступ до всіх кнопок вікна по їх дескрипторах і набуття значень тексту).

І при передачі інформації в тестований інтерфейс, і при отриманні інформації для аналізу можуть використовуватися два способи доступу до елементів інтерфейсу:

- позиційний, при якому доступ до елементу здійснюється за допомогою завдання його абсолютних (щодо екрану) або відносних (щодо вікна) координат і розмірів;
- по ідентифікатору, при якому доступ до елементу здійснюється за допомогою отримання інтерфейсного елементу за допомогою його унікального ідентифікатора в межах вікна.

При внесенні змін до призначеного для користувача інтерфейсу при використанні першого методу в результаті проведення регресійного тестування буде виявлено велику кількість не минулих тестів - достатньо зміни місцеположення одного ключового інтерфейсного елементу, як всі сценарії почнуть працювати невірно. Відповідно при такому методі автоматизації тестування необхідно міняти значну частину сценаріїв в системі тестів при кожній зміні інтерфейсу системи. Такий метод автоматизації тестування підходить для систем із сталим і рідко змінним інтерфейсом.

Другий метод автоматизації тестування стійкіший до зміни розташування інтерфейсних елементів, але зміни тестових прикладів можуть потрібно і тут у разі зміни логіки роботи інтерфейсних елементів. Наприклад, хай в першій версії системи при натисненні на кнопку "Передати дані" передача даних починалася відразу і виводилося вікно з індикатором прогресу. Сценарій тестового прикладу в цьому випадку включає імітацію натиснення на кнопку і звернення до індикатора прогресу для набуття значення прогресу у відсотках.

Якщо в другій версії системи після натиснення на кнопку "Передати дані" спочатку виводиться вікно "Ви упевнені?" з кнопками "Так і ні", то для перевірки роботи індикатора прогресу в тестовий сценарій необхідно додати імітацію натиснення кнопки "Так".

Навіть при зверненні за допомогою ідентифікаторів без модифікації тестового прикладу тест коректно не виконуватиметься. Проте, цей спосіб звернення до інтерфейсних елементів добре підходить для тестування програмних систем, в т.ч. з не сталим призначеним для користувача інтерфейсом.

6.3. Тестування зручності використання призначених для користувача інтерфейсів

Зручність використання призначеного для користувача інтерфейсу (usability) - показник його якості, який визначає кількість зусиль, необхідних для вивчення принципів роботи з програмною системою за допомогою даного інтерфейсу, її використання, підготовки вхідних даних і інтерпретації вихідних. Інакше кажучи, зручність використання визначає ступінь простоти доступу користувача до функцій системи, що надаються через людино-машинний (призначений для користувача) інтерфейс.

Тестування зручності використання призначеного для користувача інтерфейсу, взагалі кажучи, не відноситься до класичних методів тестування програмних систем. Фахівець з тестування призначеного для користувача інтерфейсу повинен поєднувати в собі знання як в області програмної інженерії, так і у фізіології, психології і ергономіці. Даний розділ курсу не претендує на повноту викладу питання і дає найзагальніші уявлення про проблематику, пов'язану з тестуванням зручності використання призначеного для користувача інтерфейсу.

На зручність використання призначеного для користувача інтерфейсу впливають наступні чинники:

- **легкість навчання** - чи швидко людина вчиться використовувати систему;
- **ефективність навчання** - чи швидко людина працює після навчання;
- **запоминаемость навчання** - чи легко запам'ятовується все, чому людина навчилася;
- **помилки** - чи часто людина допускає помилки в роботі;
- **загальна задоволеність** - чи є загальне враження від роботи з системою позитивним.

Всі ці чинники, не дивлячись на свою неформальність, можуть бути зміряні. Для таких вимірювань вибирається група типових користувачів системи, і в процесі їх роботи вимірюються показники їх роботи з системою (наприклад, кількість допущених помилок), а також їм пропонується висловити власні враження від системи за допомогою заповнення опитних листів.

Виділяють наступні етапи тестування зручності використання призначеного для користувача інтерфейсу [10].

1. **Дослідницьке** - проводиться після формулювання вимог до системи і розробки прототипу інтерфейсу. Основна мета на цьому етапі - провести високорівневе обстеження інтерфейсу і з'ясувати, чи дозволяє він з достатнім ступенем ефективності вирішувати завдання користувача.
2. **Оцінне** - проводиться після розробки низькорівневих вимог і деталізованого прототипу призначеного для користувача інтерфейсу. Оцінне тестування заглиблює дослідницьке і має ту ж мету. На даному етапі вже проводяться кількісні вимірювання характеристик призначеного для користувача інтерфейсу: вимірюються кількість звернень до системи допомоги по відношенню до кількості досконалих операцій, кількість помилкових операцій, час усунення наслідків помилкових операцій і тому подібне
3. **Валідаційне** - проводиться ближче до етапу завершення розробки. На цьому етапі проводиться аналіз відповідності інтерфейсу програмної системи стандартам, що регламентують питання зручності інтерфейсу (наприклад ISO 13407, ISO 9126), проводиться загальне тестування всіх компонент призначеного для користувача інтерфейсу з погляду кінцевого користувача. Під компонентами інтерфейсу тут розуміється як його програмна реалізація, так і система допомоги і керівництво користувача. Також на даному етапі перевіряється відсутність дефектів зручності використання інтерфейсу, виявлених на попередніх етапах.
4. **Порівняльне** - даний вид тестування може проводитися на будь-якому етапі розробки інтерфейсу. В ході порівняльного тестування порівнюються два або більш за варіанти реалізації призначеного для користувача інтерфейсу.

Як правило, при тестуванні зручності використання призначеного для користувача інтерфейсу використовуються деякі евристичні критерії і характеристики, які замінюють точні оцінки в класичному тестуванні програмних систем.

Наприклад, Якоб Нільсен в своїй роботі виділив 10 евристичних характеристик зручного призначеного для користувача інтерфейсу, які із його точки зору повинні перевірятися при тестуванні зручності використання інтерфейсу.

- **Наблюдаємость стану системи**. Система завжди повинна оповіщати користувача про те, що вона в даний момент робить, причому через розумні проміжки часу.
- **Співвідношення з реальним світом**. Термінологія, використана в інтерфейсі системи повинна співвідноситися з призначеним для користувача світом, тобто це повинні бути термінологія проблемної області користувача, а не технічна термінологія.
- **Призначене для користувача управління і свобода дій**. Користувачі часто вибирають окремі інтерфейсні елементи і використовують функції системи

помилково. В цьому випадку необхідно надавати чітко певний "аварійний вихід", за допомогою якого можна повернутися до попереднього нормального стану. До таких "аварійних виходів" відносяться, наприклад, функції відкоту і зворотного відкоту.

- **Цілісність і стандарти**. Для позначення одних і тих же об'єктів, ситуацій і дій повинні використовуватися однакові слова у всіх частинах інтерфейсу. Більш того, термінологія повідомлень в призначеному для користувача інтерфейсі повинна враховувати угоди конкретної платформи.
- **Допомога користувачам в розпізнаванні, діагностиці і усуненні помилок**. Повідомлення про помилки повинні бути написані на природній мові, а не замінюватися кодами помилок. Повідомлення про помилки повинні чітко визначати суть виниклої проблеми і пропонувати її конструктивне рішення.
- **Запобігання помилкам**. Продуманий дизайн призначеного для користувача інтерфейсу, що запобігає появі помилок користувача, завжди краще за добре продумані повідомлення про помилки. При проектуванні інтерфейсу необхідно або повністю усунути елементи, в яких можуть виникати помилки користувача, або перевіряти введення користувача в цих елементах і повідомляти його про потенційно можливе виникнення проблеми.
- **Розпізнавання, а не згадка**. При створенні інтерфейсу необхідно мінімізувати навантаження на пам'ять користувача, роблячи об'єкти, дії і опції ясними, доступними і явно видимими. Користувач не повинен запам'ятовувати інформацію при переході від одного діалогового вікна до іншого. У всіх необхідних місцях повинні бути доступні контекстні інструкції по використанню інтерфейсу.
- **Гнучкість і ефективність використання**. У інтерфейсі повинні бути передбачені гарячі клавіші (не обов'язкові до використання початкуючим користувачем) - вони часто значно прискорюють роботу досвідченого користувача. Іншими словами, система повинна надавати два способи роботи - для новачків і для досвідчених користувачів. Бажано при цьому давати можливість користувачеві автоматизувати дії, що часто повторюються.
- **Естетичний і мінімально необхідний дизайн**. Вікна не повинні містити що не відноситься до справи або рідко використовувану інформацію. Кожен інтерфейсний елемент, що містить даремну інформацію, грає роль інформаційного шуму і відволікає користувача від дійсно корисних інтерфейсних елементів.
- **Допомога і документація**. Не дивлячись на те, що в ідеальному випадку краще, коли системою можна користуватися без документації, така все одно необхідна - як у вигляді системи допомоги, так і, можливо, у вигляді друкарського керівництва. Інформація в документації повинна бути структурована так, щоб користувач міг легко знайти потрібний розділ, присвячений вирішуваному ним завданню. Кожен такий орієнтований на конкретне завдання розділ повинен окрім загальної інформації містити покрокове керівництво по виконанню завдання і не повинен бути дуже довгим.

Всі ці евристики можуть використовуватися при тестуванні зручності використання призначеного для користувача інтерфейсу. Достатньо очевидно, що при тестуванні зручності слабо застосовні способи автоматизації тестування за допомогою сценаріїв і подібні методи. Один з найбільш ефективних методів перевірки інтерфейсу на зручність - використання формальної інспекції. Питання в бланку інспекції можуть бути як загального характеру (так, наприклад, можна використовувати як питання перераховані вище 10 евристик), так і цілком конкретними. Наприклад, в роботі приводиться список контрольних питань, які бажано перевіряти при тестуванні зручності використання web-сайтів. З деякими змінами ці питання застосовні і для звичайних віконних інтерфейсів.

Контрольні запитання

1. Чому тестування є важливим етапом життєвого циклу ПЗ? Обгрунтуйте.
2. Який відсоток бюджету проекту виділяється на тестування?
3. Чим відрізняється функціональне та структурне тестування?

СПИСОК РЕКОМЕНДОВАНИХ ДЖЕРЕЛ

1. Андон Ф.И., Коваль Г.И., Коротун Т.М., Лаврищева Е.М., Суслов В.Ю Основы инженерии качества программных систем. – 2-е изд., перераб. и доп. – К.: Академпериодика, 2007. – 672 с.
2. Блэк Р. Ключевые процессы тестирования. Планирование, подготовка, проведение, совершенствование.: Пер.с англ. - М.: Изд.: Лори, 2003. – 544 с.
3. Брауде Э. Дж. Технология разработки программного обеспечения . – СПб.: Питер, 2004. – 655 с.:ил.
4. Гагарина Л.Г., Кокорева Е.В., Виснадул Б.Д. Технология разработки программного обеспечения: учебное пособие / под ред. Л.Г. Гагариной. – ИД «Форум»: ИНФРА-М, 2008. – 400 с.:ил.
5. Дастин Э., Рэшка Дж., Пол Дж. Автоматизированное тестирование программного обеспечения.: Пер.с англ.-М.: Изд-во: Лори, 2003. – 592 с.
6. Канер С., Фолк Дж., Нгуен Е.К. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений: Пер. с англ. – К.: Издательство «Диасофт», 2001. – 544 с.
7. Лавріщева К.М. Програмна інженерія.–К.– 2008.–319 с.
8. Лаврищева Е.М., Петрухин В.А. Методы и средства инженерии программного обеспечения: Учебник. – М.: МФТИ(ГУ), 2003. – 304 с.
9. Соммервилл И. Инженерия программного обеспечения, 6-е изд.: Пер. с англ. – М.: Вильямс, 2002. – 624 с.: ил.
10. Тамре Л. Введение в тестирование программного обеспечения.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. - 368 с.
11. Шафер Д., Фарелл Р., Шафер А. Управление программными проектами: достижение оптимального качества при минимуме затрат.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. - 1136 с.
12. Матеріали по тестуванню ПЗ. – Режим доступу: http://www.znannya.org/?view=software-testing
13. ДСТУ 2850-94. Програмні засоби ЕОМ. Показники і методи оцінювання якості
14. ДСТУ 2462-94. Сертифікація. Основні поняття, терміни та визначення.
15. Калбертсон Р., Браун К., Кобб Г. Быстрое тестирование.: Пер. с англ. – М.: Издательство: Вильямс, 2002. - 384 с.
16. Липаев В.В. Обеспечение качества программных средств. Методы и стандарты. Серия «Информационные технологии». – М.: СИНТЕГ, 2001.- 380 с.
17. Макгрегор Дж., Сайкс Д. Тестирование объектно-ориентированного программного обеспечения. Практическое пособие. - К.: DiaSoft, 2002. - 432 с.