Лекція 3.
Класифікація видів тестування програмного забезпечення

Зміст

Цільова настанова: Розглядається класифікація видів тестування за кількома ознаками, а саме: по об'єкту тестування, за повнотою інформації про об'єкт тестування, за ступенем автоматизації процесу тестування, за ступенем ізольованості компонентів, за часом проведення тестування, за рівнем тестування, по стратегії тестування, за ознакою позитивності сценаріїв , за ступенем підготовленості до тестування.

Ключові слова: класифікація видів тестування, об'єкту тестування, повнота інформації, ступінь автоматизації процесу тестування, ступінь ізольованості компонентів, часу проведення тестування, рівень тестування, позитивність сценаріїв, ступінь підготовленості до тестування.

1. Класифікація видів тестування

Існує кілька ознак, за якими прийнято проводити класифікацію видів тестування. Зазвичай виділяють наступні:

1.1. По об'єкту тестування:

- **функціональне тестування**

- це тестування ПО з метою перевірки можливості бути реалізованим функціональних вимог , тобто здатності ПО в певних умовах вирішувати завдання, необхідні користувачам (функціональні вимоги визначають, що саме робить ПЗ, які завдання воно вирішує).

Функціональні вимоги включають:

- функціональна придатність;
- точність;
- здатність до взаємодії;
- відповідність стандартам і правилам;
- захищеність.

- **тестування продуктивності**

- це тестування ПО, яке проводиться з метою визначення, як швидко працює система або її частина під певним навантаженням. Також може служити для перевірки і підтвердження інших атрибутів якості системи, таких як м асштабіруемость (здатність системи збільшувати свою продуктивність при додаванні ресурсів (зазвичай апаратних)), надійність і споживання ресурсів. Тестування продуктивності - це одна зі сфер діяльності, яка прагне враховувати продуктивність на стадії моделювання та проектування системи, перед початком основної стадії кодування.

У загальному випадку тестування продуктивності може проводитися з різними цілями:

- з метою демонстрації того, що система задовольняє критеріям продуктивності.
- з метою з'ясування, у який із двох або декількох систем продуктивність краще.
- з метою визначення, який елемент навантаження або частина системи призводить до зниження продуктивності.

Багато тестів на продуктивність робляться без спроби осмислити їх реальні цілі. Перед початком тестування завжди повинен бути заданий бізнес-питання: «Яку мету ми переслідуємо, тестуючи продуктивність?». Відповіді на це питання є частиною техніко-економічного обгрунтування тестування. Цілі можуть відрізнятися в залежності від технологій, які використовуються додатком, або його призначення, проте, вони завжди включають щось з нижченаведеного:

- *Паралелізм / Пропускна здатність* - це максимальне число паралельних працюючих користувачів додатка, підтримка якого очікується від програми в будь-який момент часу.
- *Час відповіді сервера* - ця концепція будується навколо часу відповіді одного вузла додатки на запит, надісланий іншим вузлом.
- *Час відображення* - одне з найскладніших для додатка для навантажувального тестування понять, так як в загальному випадку вони не використовують концепцію роботи з тим, що відбувається на окремих вузлах системи, обмежуючись тільки розпізнаванням періоду часу протягом якого немає мережевої активності. Для того, щоб заміряти час відоб-

раження, в загальному випадку потрібно включати функціональні тестові сценарії в тести продуктивності, але більшість програм для тестування продуктивності не включають в себе таку можливість.

Типові питання тестування продуктивності:

1. Що охоплюється тестом продуктивності? Які підсистеми, компоненти, інтерфейси і т.д. повинні бути протестовані?
2. Якщо в тест включаються інтерфейси, то скільки одночасно працюючих в системі користувачів очікується для кожного інтерфейсу (необхідно визначити пікові і нормальні значення)
3. Як виглядає апаратна складова тестованої системи? (Необхідно описати все сервера та мережеве обладнання)
4. Який сценарій використання кожної компоненти системи? (наприклад, 20% запитів становить вхід в систему, 40% - пошук, 30% - вибір елемента, 10% - вихід із системи)
5. Який сценарій використання системи? (в одному тесті на продуктивність можуть бути задіяні різні сценарії використання кожного компонента)
6. Які вимоги до часу виконання серії операцій серверної частини програми?

У тестуванні продуктивності розрізняють наступні напрямки:

- навантажувальний;
- стрес;
- тестування стабільності;
- конфигурационное.

Розглянемо ці напрямки докладніше.

Тестування навантаження - визначення або збір показників продуктивності і часу відгуку програмно-технічної системи або пристрої у відповідь на зовнішній запит (вплив) з метою встановлення відповідності вимогам, що пред'являються до даної системи (пристрою), причому створювана навантаження на систему не перевищує нормальні сценарії її використання . Термін тестування навантаження може бути використаний в різних значеннях в професійному середовищі тестування ПО. У загальному випадку він означає практику моделювання очікуваного використання додатка за допомогою емуляції роботи декількох користувачів одночасно. Таким чином, подібне тестування найбільше підходить для мультіпользовательской систем, частіше - використовують клієнт-серверну архітектуру (наприклад, веб-серверів). Однак і інші типи систем ПО можуть бути протестовані подібним способом. Наприклад, текстовий або графічний редактор можна змусити прочитати дуже великий документ; а фінансовий пакет - згенерувати звіт на основі даних за кілька років. Найбільш адекватно спроектований навантажувальний тест дає більш точні результати. Основна мета навантажувального тестування полягає в тому, щоб, створивши певну очікувану в системі навантаження (наприклад, за допомогою віртуальних користувачів) і використавши ідентичне програмне і апаратне забезпечення, спостерігати за показниками продуктивності системи. В ідеальному випадку в якості критеріїв успішності навантажувального тестування ви-

ступають вимоги до продуктивності системи, які формулюються і документуються на стадії розробки функціональних вимог до системи до початку програмування основних архітектурних рішень. Однак часто буває так, що такі вимоги не були чітко сформульовані або були сформульовані зовсім. В цьому випадку перше тестування навантаження буде пробним і грунтуватися на розумних припущеннях про очікувану навантаженні і споживанні апаратної частини ресурсів. Одним з оптимальних підходів у використанні навантажувального тестування для вимірювань продуктивності системи є тестування на стадії ранньої розробки. Тестування навантаження на перших стадіях готовності архітектурного рішення з метою визначити його спроможність називається «Proof - of - Concept» тестуванням.

Стрес-тестування - один з видів тестування ПО , яке оцінює надійність і стійкість системи в умовах перевищення меж нормального функціонування. Стрес-тестування особливо необхідне для «критично важливого» ПО, однак також використовується і для решти ПО. Зазвичай стрес-тестування краще виявляє стійкість, доступність і обробку винятків системою під великим навантаженням, ніж те, що вважається коректною поведінкою в нормальних умовах. Термін «стрес-тестування» часто помилково использу ют як синонім навантажувального тестування, а також тестування продуктивності . Це невірно, тому що ці види тестування відповідають на різні бізнес-питання і використовують різну методологію.

У загальному випадку методологія стрес-тестування заснована на фіксації і аналізі показників продуктивності додатка при навантаженнях, що перевищують очікувані на стадії супроводу, і має на меті визначити стійкість додатки на випадок сплеску активності по його використанню.

Необхідність стрес-тестування диктується наступними факторами:

- Велика частина всіх систем розробляються з допущенням про функціонування в нормальному режимі і навіть в разі, коли допускається можливість збільшення навантаження, реальні обсяги її збільшення не беруться до уваги.
- У разі контракта- угоди про рівень послуг вартість відмови системи в екстремальних умовах може бути дуже велика.
- Виявлення деяких помилок або дефектів у функціонуванні системи часто можливо з використанням інших типів тестування.
- Тестування, проведеного розробником, може бути недостатньо для емуляції умов, при яких відбувається відмова системи.
- Переважно бути готовим до обробки екстремальних умов системи, ніж очікувати її відмови.

Основні напрямки застосування стрес-тестування:

1. Загальне дослідження поведінки системи при пікових навантаженнях.
2. Дослідження обробки помилок і виняткових ситуацій системою при пікових навантаженнях.
3. Дослідження вузьких місць системи або окремих компонент при диспропорційних навантаженнях.

4. Тестування ємності системи - є одним з найважливіших з точки зору розвитку бізнесу напрямків стрес-тестування і найскладніших з точки зору проведення і аналізу. Тестування Ємності проводиться з метою визначити запас міцності системи при повній відповідності вимогам до продуктивності. При моделюванні навантаження для тестування ємності системи враховується як поточна навантаження у вигляді кількості та пропорцій одночасно надходять в систему запитів, так і очікувана в перспективі. Результатом тестування ємності додатки або системи є набір максимально допустимих показників навантаження системи, при яких додаток або система відповідає вимогам до продуктивності, розробленим і документованим на етапі проектування архітектури.

Стрес-тестування, як і тестування навантаження, також може бути використано для регулярної оцінки змін продуктивності з метою отримання для подальшого аналізу динаміки зміни поведінки системи за тривалий період.

При стрес-тестуванні можливе використання п ропорціональной і диспропорційною навантаження.

Пропорційне навантаження. Стрес-тестування може застосовуватися як для відокремлених додатків, так і для розподілених систем з клієнт-серверною архітектурою. Найпростішим прикладом стрес-тестування відокремленого програми може бути відкриття файлу розміром в 50 Мб програмою Notepad, яка входить в комплект ОС Windows. Умови стрес-тестування програми зазвичай формуються виходячи з критичних бізнес-процесів його функціональності, обумовленими на стадії розробки вимог і аналізу ризиків. У загальному випадку в якості умов для стрес-тестування може використовуватися лінійно збільшена очікувана навантаження.

Диспропорційно навантаження. У разі тестування багатоланкових розподілених систем необхідно враховувати вже не тільки фактичний обсяг навантаження, що складається з безлічі елементів, але і їх пропорції в загальному обсязі. Використання диспропорційною навантаження в стрес-тестах може також застосовуватися для виявлення вузьких місць окремих компонентів системи.

Тестування стабільності проводиться з метою переконатися в тому, що додаток витримує очікуване навантаження протягом тривалого часу. При проведенні цього виду тестування здійснюється спостереження за споживанням додатком пам'яті, щоб виявити потенційні витоку. Також важливим, але часто непоміченим, фактором є деградація продуктивності, сенс якого зводиться до того, щоб швидкість обробки інформації і / або час відповіді додатки через тривалий час роботи були такими ж або краще, ніж на самому початку тесту.

Конфігураційне тестування - це ще один з видів традиційного тестування продуктивності. У цьому випадку замість того, щоб тестувати продуктивність системи з точки зору подається навантаження, тестується ефект впливу на продуктивність змін в конфігурації. Хорошим прикладом такого тестування можуть бути експерименти з різними методами балансування навантаження. Конфігураційне тестування також може бути поєднане з навантажувальним, стрес або тестуванням стабільності.

- **Тестування зручності використання** (юзабіліті-тестування) - експеримент, що виконується з метою визначення, наскільки добре люди можуть використовувати певний штучний об'єкт (такий як веб-сторінка , призначений для користувача інтерфейс або пристрій) для його передбачуваного застосування, тобто юзабіліті-тестування вимірює ергономічність об'єкта. Юзабіліті-тестування - метод оцінки зручності продукту у використанні, заснований на залученні користувачів в якості тестувальників, випробувачів і підсумовуванні отриманих від них висновків. Існує ще один підхід до юзабіліті-тестування: для вирішення завдання запропонованої користувачеві розробляється «ідеальний» сценарій вирішення цього завдання. Як правило, це сценарій, на який орієнтувався розробник. При виконанні завдання користувачами реєструються їх відхилення від задуманого сценарію для подальшого аналізу. Після декількох ітерацій доопрацювання програми і подальшого тестування можна отримати задовільний з точки зору користувача інтерфейс.
- **Тестування інтерфейсу користувача** - являє собою сукупність засобів і методів, за допомогою яких користувач взаємодіє з різними, найчастіше складними, елементами, машинами і пристроями. Інтерфейс користувача - це різновид інтерфейсів, в якому одна сторона представлена людиною (користувачем), інша - машиною / пристроєм. інтерфейс двонаправлений - пристрій, отримавши команди від користувача і виконавши їх, видає інформацію назад, наявними у нього засобами (візуальними, звуковими і т. п.), прийнявши яку, користувач видає пристрою наступні команди наданими в його розпорядження коштами (кнопки, перемикачі, регулятори, сенсори, голосом, і т. д.). Найчастіше термін застосовується по відношенню до комп'ютерних програм, однак під ним може матися на увазі будь-яка система взаємодії з пристроями, здатними до інтерактивного спілкування з користувачем.

Інтерфейс користувача комп'ютерної програми включає:

  - засоби відображення інформації, що відображається інформацію, формати і коди ;
  - командні режими, мова «користувач - інтерфейс»;
  - пристрою і технології введення даних;
  - діалоги, взаємодія і транзакції між користувачем і комп'ютером, зворотний зв'язок з користувачем;
  - підтримку прийняття рішень у конкретній предметній області ;
  - порядок використання програми і документацію на неї.
- **Тестування безпеки** - оцінка уразливості програмного забезпечення до різних атак. Комп'ютерні системи дуже часто є мішенню незаконного проникнення. Під проникненням розуміється широкий діапазон дій: спроби хакерів проникнути в систему з спортивного інтересу, помста розгніваних службовців, злом шахраями для незаконної наживи. Тестування безпеки перевіряє фактичну реакцію захисних механізмів, вбудованих в систему, на проникнення. В ході тестування безпеки випробувач грає роль зломщика.

Йому дозволено все:

  - спроби дізнатися пароль за допомогою зовнішніх засобів;
  - атака системи за допомогою спеціальних утиліт, які аналізують захисту;

- придушення, приголомшення системи (в надії, що вона відмовиться обслуговувати інших клієнтів);
- цілеспрямоване введення помилок в надії проникнути в систему в ході відновлення;
- перегляд несекретних даних в надії знайти ключ для входу в систему.

Звичайно, при необмеженому часі і ресурсах хороше тестування безпеки зламає будь-яку систему. Завдання проектувальника системи - зробити ціну проникнення вищою, ніж ціна одержуваної в результаті інформації.

- **Тестування локалізації** - це перевірка коректності перекладу, текст повинен бути граматично, синтаксично і логічно правильним. Локалізації вимагає рішення кілька завдань і питань:
  - робити локалізацію власними силами або залучати аутсорсинг або комбінувати власні сили з аутсорсингом.
  - визначити список підтримуваних мов.
  - використовувати чи ні псевдолокалізацію як підготовчий етап локалізації (псевдо-локалізація - це створення продукту неіснуючою мовою. Ця мова ідентичний англійському з тією лише різницею, що в ньому кожна буква замінена символом, зовні нагадує її накреслення).
  - здійснити переклад користувацького інтерфейсу.
  - здійснити переклад системних повідомлень і помилок.
  - здійснити переклад help і супутньої документації.
  - перевірка правильності перекладу в контексті цього додатка.
  - перевірка локалізованих додатків на різних мовних платформах.

Починати тестування переважно зі статичних елементів, перевірки написів на статичних елементах: заголовки блоків, пояснюючі написи і т.п., саме на них користувач буде звертати увагу в першу чергу. Перевіряючи подібні елементи, не забувайте, що довжина одного і того ж тексту на різних мовах може істотно відрізнятися. Відповідно, постарайтеся переконатися в тому, щоб для всіх необхідних написів знайшлося місце в розмітці вашого сайту або програми. Після того, як зі статичними елементами буде покінчено, переходите до решти контроль : кнопок, меню і т.п. Пам'ятайте, що в залежності від реалізації, локалізація елементів управління може бути визначена в коді, а може залежати від налаштувань вашого браузера або ОС. Не забувайте і про повідомлення про помилки. Плануйте тестування і складайте тест кейси таким чином, щоб перевірити максимальну кількість повідомлень про помилки. Програмісти чомусь приділяють дуже мало уваги подібним речам, через що значна частина повідомлень про помилки може бути взагалі не внесена в локалізовану версію або ж бути не перекладеної на потрібну мову, або бути абсолютно нечитабельною через проблеми з кодуванням. Переконайтеся, що введення даних може бути здійснений на мові локалізації. Якщо тестоване веб додаток передбачає введення будь-яких даних користувачем, то обов'язково переконайтеся в тому, що користувачі мають можливість вводити дані на мові локалізації і що всі символи національного алфавіту, введені користувачем, відображаються і обробляються додатком коректно.

Ще один важливий момент, на який слід звернути увагу при тестуванні - регіональні і національні особливості країни, для якої призначена локалізація. До таких особливостей можна віднести напрям тексту, формати дат, адрес, десяткових дробів, позначення валют, одиниці вимірювання різних величин і т.д.

- Тестування сумісності - метод, основною метою якого є забезпечення якісної роботи кінцевого продукту з іншим програмним забезпеченням. Наприклад, велика кількість браузерів, що використовують різні движки, змушують веб-розробників шукати різноманітні способи перевірки свого сайту на сумісність з ними.

1.2. По повноті інформації про об'єкт тестування:

- **Тестування чорного ящика**

У цій стратегії програма розглядається як чорний ящик. Метою тестування ставиться з'ясування обставин, в яких поведінка програми не відповідає специфікації. Для виявлення всіх помилок в програмі необхідно виконати *вичерпне тестування*, тобто тестування на всіляких наборах даних. Для більшості програм це неможливо, тому застосовують розумне тестування , при якому тестування програми обмежується невеликим підмножиною всіляких наборів даних. При цьому необхідно вибирати найбільш підходящі підмножини, підмножини з найвищою імовірністю виявлення помилок.

Стандартна процедура тестування «чорним ящиком» включає в себе наступну послідовність подій:

1. *Планування* - визначається стратегія, розробляють серії тестів, розподіляються завдання між співробітниками, етап планування слід відразу за розробкою вимог.

2. *Приймальне (кваліфікаційне) тестування* - це тестування, що проводиться під час вступу кожної нової версії ПП (перевірка на стабільність). Якщо ПП таку перевірку не проходить, то такий продукт далі тестувати недоцільно, і він повертається розробником.

3. *Функціональне та системне тестування* , звірка і атестація продукту - на цьому етапі продукт звіряється з проектними документами. Зокрема, проводиться функціональне тестування, в рамках якого продукт звіряється зі специфікацією, потім проводиться тестування цілісності (призначені для користувача вимоги) і системне тестування (системні вимоги). Тестування цілісності і системне тестування звуться атестаційного тестування.

4. *Навантажувальні випробування*. При навантажувальних випробуваннях перевіряється реакція програми на граничні умови експлуатації:

• тестування на максимальний обсяг даних;

• тестування на максимальну кількість одночасних запитів (для web - додатків);

• аналіз вимог до ресурсів (ОЗУ і дисковий простір).

5. *α і β, реліз - тестування*.

Методи стратегії чорного ящика:

1. Еквівалентну розбиття . Розробка тестів цим методом здійснюється в два етапи : виділення класів еквівалентності і побудова тесту.

Основу методу складають два положення:

- Вихідні дані необхідно розбити на кінцеве число класів еквівалентності. В одному класі еквівалентності містяться такі тести, що, якщо один тест з класу еквівалентності виявляє деяку помилку, то і будь-який інший тест з цього класу еквівалентності повинен виявляти цю саму помилку.
- Кожен тест повинен включати, по можливості, якомога більше класів еквівалентності, щоб мінімізувати загальне число тестів.

2. Аналіз граничних значень відрізняється від еквівалентного роздроблення наступним:
   - Вибір будь-якого елементу в класі еквівалентності в якості представницького здійснюється таким чином, щоб перевірити тестом кожну кордон цього класу.
   - При розробці тестів розглядаються не тільки вхідні значення (простір входів), але і вихідні (простір виходів).

Метод вимагає певною мірою творчості і наявності спеціалізації. Існує кілька правил:

- Побудувати тести з неправильними вхідними даними для ситуації незначного виходу за межі області значень.
- Обов'язково писати тести для мінімальної і максимальної межі діапазону.
- Використовувати перші два правила для кожного з вхідних значень (використовувати пункт 2 для всіх вихідних значень).
- Якщо вхід і вихід програми представляє впорядкована множина, зосередити увагу на першому і останньому елементах списку.

Аналіз граничних значень, якщо він застосований правильно, дозволяє виявити велику кількість помилок. Однак визначення цих кордонів для кожного завдання може бути окремою важким завданням. Також цей метод не перевіряє комбінації вхідних значень.

3. *Аналіз причинно-наслідкових зв'язків*. Етапи побудови тесту:
   - Специфікація розбивається на робочі ділянки.
   - У специфікації визначаються безліч причин і наслідків. Під причиною розуміється окрема вхідна умова або клас еквівалентності. Слідство є вихідна умова або перетворення системи. Тут кожну причину і слідству присвоюється номер.
   - На основі аналізу семантичного (смислового) змісту специфікації будується таблиця істинності, в якій послідовно перебираються всілякі комбінації причин і визначаються наслідки для кожної комбінації причин.

Таблиця забезпечується примітками, які задають обмеження і описують комбінації, які неможливі. Недоліком цього підходу є погане дослідження граничних умов.

4. *Припущення про помилку*. Програміст з великим досвідом вишукує помилки без всяких методів, але при цьому він підсвідомо використовує метод припущення про помилку. Даний метод в значній мірі заснований на інту-

їції. Основна ідея методу полягає в тому, щоб скласти список, який перераховує можливі помилки і ситуації, в яких ці помилки могли проявитися. Потім на основі списку складаються тести.

- **Тестування білого ящика**, також звана стратегією тестування керована логікою програми дозволяє перевірити внутрішню структуру програми. Виходячи з цієї стратегії тестувальник отримує тестові дані шляхом аналізу логіки роботи програми.

Стратегія Білого ящика включає в себе наступні методи тестування:

1. *покриття операторів*

Критерії покриття операторів на увазі виконання кожного оператора програми, по крайней мере, один раз.

2. *покриття рішень*

Відповідно до цього критерію необхідно скласти таке число тестів, при яких кожна умова в програмі візьме як істинне значення, так і хибне значення.

3. *покриття умов*

Цей критерій є більш ефективним в порівнянні з попередніми.

Записується число тестів достатню для того, щоб всі можливі результати кожної умови в рішенні були виконані, принаймні, один раз. Однак цей критерій не завжди призводить до виконання кожного оператора, по крайней мере, один раз. Тому до цього критерію додається додаткова умова, кожен оператор повинен бути виконаний хоча б один раз. Якщо після складання тестів у нас залишаться не покриті оператори, то ми повинні доповнити свій набір тестів таким чином, щоб кожен оператор виконується не менше одного разу.

4. *покриття рішень і умов*

Відповідно до цього критерію необхідно скласти тести так, щоб результати кожного умови виконувалися хоча б один раз, результати кожного рішення так само виконувалися хоча б один раз, і кожен оператор повинен бути виконаний хоча б один раз. Хоча метод і є досить потужним і дозволяє знаходити досить велика кількість помилок, він має і недоліки:

- не завжди можна перевірити всі умови;
- неможливо перевірити умови, які приховані іншими умовами;
- метод володіє недостатньою чутливістю до помилок в логічних виразах.

5. *комбинаторное покриття умов*

Це критерій вимагає, щоб всі можливі комбінації результатів умов в кожному рішенні , а також кожен оператор виконалися , по крайней мере, один раз. Таким чином, для програм, що містять тільки одну умову на кожне рішення, мінімальним є критерій набору тестів якого:

- викликає виконання всіх результатів кожного рішення, принаймні, один раз;
- виконує кожен оператор, по крайней мере, один раз.

Для програм входить більш як один умови мінімальний критерій складається з набору тестів, що викликають виконання всіх можливих комбінацій результатів умов і виконує кожен оператор мінімум один раз.

1.3. За ступенем автоматизації процесу тестування:

- **Ручне тестування** - проведення тестів при прямому (безпосередньому) участі людини, як правило, протягом тривалого часу, що відрізняється суб'єктивністю при прийнятті рішень.
- **Автоматизоване тестування** використовує програмні засоби для виконання тестів і перевірки результатів виконання, що допомагає скоротити час тестування і спростити його процес. Існує два основні підходи до автоматизації тестування: тестування на рівні коду і GUI-тестування.

  До першого типу відносяться, зокрема, модульне тестування .

  До другого - імітація дій користувача за допомогою спеціальних тестових фреймворків.

  Найбільш поширеною формою автоматизації є тестування додатків через графічний користувальницький інтерфейс. Популярність такого виду тестування пояснюється двома факторами: по-перше, додаток тестується тим же способом, яким його буде використовувати людина, по-друге, можна тестувати додаток, не маючи при цьому доступу до вихідного коду.

  Однією з головних проблем автоматизованого тестування є його трудомісткість: незважаючи на те, що воно дозволяє усунути частина рутинних операцій і прискорити виконання тестів, великі ресурси можуть витрачатися на оновлення самих тестів. Це відноситься до обох видів автоматизації.

  Часто буває необхідно оновити та модульні тести, і зміна коду тестів може зайняти стільки ж часу, скільки і зміна основного коду. З іншого боку, при зміні інтерфейсу програми необхідно заново переписати всі тести, які пов'язані з оновленими вікнами, що при великій кількості тестів може відняти значні ресурси.

  Для автоматизації тестування існує велика кількість додатків. Найбільш популярні:
  - HP QuickTest Professional, HP LoadRunner, HP Quality Center
  - Segue SilkPerformer
  - IBM Rational FunctionalTester, IBM Rational PerformanceTester, IBM Rational TestStudio
  - AutomatedQA TestComplete

  Використання цих інструментів допомагає тестувальникам автоматизувати наступні завдання:
  - установка продукту;
  - створення тестових даних;
  - GUI взаємодія;
  - визначення проблеми.

  Однак автоматичні тести не можуть повністю замінити ручне тестування. Автоматизація всіх випробувань - дуже дорогий процес, і тому автоматичне тестування є лише доповненням ручного тестування. Найкращий варіант використання автоматичних тестів - регресійні тестування.
- **Напівавтоматизованого тестування** - комбінований метод.

1.4. За ступенем ізольованості компонентів:

- **Компонентний (модульне) тестування** або юніт-тестування - процес, що дозволяє перевірити на коректність окремі модулі вихідного коду програми. Ідея полягає в тому, щоб писати тести для кожної нетривіальною функції або методу. Це дозволяє досить швидко перевірити, чи не призвело чергову зміну коду до регресії , тобто до появи помилок в уже оттестировать місцях програми, а також полегшує виявлення і усунення таких помилок.

Мета модульного тестування - ізолювати окремі частини програми і показати, що окремо ці частини працездатні. Цей тип тестування зазвичай виконується програмістами .

Переваги цього методу:

- *Заохочення змін*. Модульне тестування дозволяє програмістам проводити рефакторинг (процес зміни внутрішньої структури програми, що не зачіпає її зовнішньої поведінки і має на меті полегшити розуміння її роботи), будучи впевненими, що модуль як і раніше працює коректно (регресійні тестування). Це заохочує програмістів до змін коду, оскільки досить легко перевірити, що код працює і після змін.
- *Спрощення інтеграції*. Модульне тестування допомагає усунути сумніви з приводу окремих модулів і може бути використано для підходу до тестування «знизу вгору»: спочатку тестуються окремі частини програми, потім програма в цілому.
- *Документування коду*. Модульні тести можна розглядати як «живий документ» для тестованого класу .
- *Відділення інтерфейсу від реалізації*. Оскільки деякі класи можуть використовувати інші класи, тестування окремого класу часто поширюється на пов'язані з ним класи. Наприклад, клас користується базою даних; в ході написання тесту програміст виявляє, що тесту доводиться взаємодіяти з базою. Це помилка, оскільки тест не повинен виходити за кордон класу. В результаті розробник абстрагується від з'єднання з базою даних і реалізує цей інтерфейс, використовуючи свій власний об'єкт -заглушку. Це призводить до менш пов'язаному коду, мінімізуючи залежності в системі.
- *Додатки модульного тестування* - є кстремальное, яке передбачає як один з постулатів використання інструментів автоматичного модульного тестування. Цей інструментарій може бути створений або третьою стороною, або групою розробників цього додатка. В екстремальному програмуванні використовуються модульні тести для розробки через тестування. Для цього розробник до написання коду пише тести, що відображають вимоги до модуля. Очевидно, жоден з цих тестів до написання коду працювати не повинен. Подальший процес зводиться до написання найкоротшого коду, який задовольняє даному набору тестів.

- **Інтеграційне тестування** - це одна з фаз тестування програмного забезпечення, при якому окремі програмні модулі об'єднуються і тестуються в групі. Зазвичай інтеграційне тестування проводиться після модульного тестування і передує системному тестуванню.

Інтеграційне тестування в якості вхідних даних використовує модулі, над якими було проведено модульне тестування, групує їх в більш великі безлічі, виконує тести, певні в плані тестування для цих множин, і представляє їх в якості вихідних даних і вхідних для подальшого системного тестування.
Метою інтеграційного тестування є перевірка відповідності проектованих одиниць функціональним, прийомним і вимогам надійності. Тестування цих проектованих одиниць - об'єднання, безлічі або група модулів - виконуються через їх інтерфейс, використовуючи тестування «чорного ящика».

Для автоматизації інтеграційного тестування застосовуються системи безперервної інтеграції (CIS). Принцип дії таких систем полягає в наступному:

1. CIS проводить моніторинг системи контролю версій;
2. При зміні вихідних кодів в репозитарії (сховище даних) проводиться оновлення локального сховища;
3. Виконуються необхідні перевірки і модульні тести;
4. Тексти програм компілюються в готові виконувані модулі;
5. Виконуються тести інтеграційного рівня;
6. Генерується звіт про тестування.

Таким чином, автоматичні інтеграційні тести виконуються відразу ж після внесення змін, що дозволяє виявляти і усувати помилки в короткі терміни.

- **Системне тестування** - це тестування програмного забезпечення (ПО), що виконується на повної, інтегрованій системі, з метою перевірки відповідності системи вихідним вимогам. Системне тестування відноситься до методів тестування чорного ящика, і, тим самим, не вимагає знань про внутрішній устрій системи.

Основним завданням системного тестування є перевірка як функціональних, так і не функціональних вимог в системі в цілому. При цьому виявляються дефекти, такі як неправильне використання ресурсів системи, непередбачені комбінації даних користувача рівня, несумісність з оточенням, непередбачені сценарії використання, відсутня або неправильна функціональність, незручність використання і т.д. Для мінімізації ризиків, пов'язаних з особливостями поведінки в системи в тому чи іншому середовищі, під час тестування рекомендується використовувати оточення максимально наближене до того, на яке буде встановлено продукт після видачі.
Можна виділити два підходи до системного тестування:

1. на базі вимог (для кожного вимоги пишуться тестові випадки (тест-кейси), перевіряючі виконання даної вимоги);
2. на базі випадків використання (альфа-тестування і бета-тестування є підкатегоріями системного тестування).

1.5. За часом проведення тестування:

- **Альфа-тестування** - використання незавершеної (альфа) версії програмного продукту, в якій реалізована не вся функціональність, запланована для даної версії продукту, штатними програмістами (розробниками і тестерами) з метою виявлення помилок в роботі реалізованих модулів і функцій для їх подальшого усунення.

- *Тестування при прийманні (димове тестування)* означає мінімальний набір тестів на явні помилки. Димової тест зазвичай виконується самим програмістом; що не проходить цей тест програму не має сенсу віддавати на більш глибоке тестування. Приклад: Помилки інсталяції: якщо програмний продукт не встановлюється, його тестування, швидше за все, виявиться неможливим.
- *Тестування нової функціональності*.
- *Регресійне тестування* - збірна назва для видів тестування програмного забезпечення, спрямованих на виявлення помилок в уже протестованих ділянках вихідного коду. Такі помилки - коли після внесення змін до програми перестає працювати то, що раніше працювало, - називають *регресійний помилками*.
- Зазвичай використовуються методи регресійного тестування включають повторні прогони попередніх тестів, а також перевірки, чи не потрапили регресивні помилки в чергову версію в результаті злиття коду.
- З досвіду розробки ПО відомо, що повторна поява однієї і тих же помилок - випадок досить частий. Тому вважається хорошою практикою при виправленні помилки створити тест на неї і регулярно проганяти його при подальших змінах програми. Хоча регресійне тестування може бути виконано і вручну, але частіше за все це робиться за допомогою спеціалізованих програм, що дозволяють виконувати всі регресійні тести автоматично. У деяких проектах навіть використовуються інструменти для автоматичного прогону регресійних тестів через заданий інтервал часу. Зазвичай це виконується після кожної вдалої компіляції (в невеликих проектах) або кожен день або кожну тиждень.
- *Тестування при здачі*.

- **Бета-тестування** - інтенсивне використання майже готової версії продукту (як правило, програмного або апаратного забезпечення) з метою виявлення максимального числа помилок в його роботі для їх подальшого усунення перед остаточним виходом (релізом) продукту на ринок, до масового споживача. На відміну від альфа-тестування, проведеного силами штатних розробників або тестувальників, бета-тестування передбачає залучення добровольців з числа звичайних майбутніх користувачів продукту, яким доступна згадана попередня версія продукту (так звана бета-версія). Такими добровольцями (їх називають бета-тестерами) часто рухає цікавість до нового продукту - цікавість, заради задоволення якого вони цілком згодні миритися з можливістю випробувати наслідки ще не знайдених (а тому і не виправлених) помилок. Крім цікавості, мотивація може бути обумовлена почуттям власної важливості від відчуття участі у важливій справі, бажанням вплинути на процес розробки і в підсумку отримувати більш задовольняє їхні потреби продукт і багатьом іншим. Крім того, відкриття бета-тестування може використовуватися як частина стратегії просування продукту на ринок (наприклад, безкоштовна роздача бета-версій дозволяє залучити широку увагу споживачів до остаточної дорогої версії продукту), а також для отримання попередніх відгуків про нього від широкого кола майбутніх користувачів. Бета-версія не є фінальною

версією продукту, тому розробник не гарантує повної відсутності помилок,які можуть порушити роботу комп'ютера і / або привести до втрати даних. Хоча, і в фінальних версіях все частіше таких гарантій розробники не дають.

- **Реліз** - випуск остаточної версії програми - готового для використання продукту. У релізі зазвичай збирають всі версії і оновлення, і випускають кінцевий продукт з усіма виправленнями, який не потрібно оновлювати, так як він є найновішою версією. Метою процесу управління релізами є консолідація (логічне об'єднання), структурування і оптимізація всіх змін або оновлень, а також зниження ризику при перекладі сервісу на новий якісний рівень. Для досягнення поставленої мети необхідно правильно розподілити наявні ресурси і розрахувати баланс між термінами, які відведені для впровадження всіх оновлень, і часом, необхідним для підготовки внесення даних змін. Процес управління релізами складається з трьох етапів:

• 1 етап: може, не завжди бути застосовний в тій чи іншій організації, але для деяких компаній, даний етап може бути одним з основоположних, до них можуть ставитися, наприклад, компанії з розробки програмних засобів або конструкторські бюро;

• 2 етап: на даному етапі необхідно визначити спочатку критерії, за якими буде проводитися тестування для кожного релізу, тобто визначити ступінь визначення готовності релізу до поширення і впровадження.

• 3 етап: поширення і впровадження.

1.6. За рівнем тестування:

- **Модульне тестування** (юніт-тестування) - тестується мінімально можливий для тестування компонент, наприклад, окремий клас або функція. Часто модульне тестування здійснюється розробниками ПЗ.
- **Інтеграційне тестування** - тестуються інтерфейси між компонентами, підсистемами. При наявності резерву часу на даній стадії тестування ведеться ітераційно, з поступовим підключенням наступних підсистем.
- **Системне тестування** - тестується інтегрована система на її відповідність вимогам.

•*Альфа-тестування* - імітація реальної роботи з системою штатними розробниками, або реальна робота з системою потенційними користувачами / замовником. Найчастіше альфа-тестування проводиться на ранній стадії розробки продукту, але в деяких випадках може застосовуватися для закінченого продукту в якості внутрішнього приймального тестування. Іноді альфа-тестування виконується під отладчиком або з використанням оточення, яке допомагає швидко виявляти знайдені помилки. Виявлені помилки можуть бути передані тестувальникам для додаткового дослідження в оточенні, подібному тому, в якому буде використовуватися програма.

•*Бета-тестування* - в деяких випадках виконується поширення версії з обмеженнями (по функціональності або часу роботи) для певної групи осіб, з тим, щоб переконатися, що продукт містить досить мало помилок. Іноді бета-тестування виконується для того, щоб отримати зворотній зв'язок про продукт від його майбутніх користувачів. Часто для вільного / відкритого ПЗ стадія

альфа-тестування характеризує функціональне наповнення коду, а бета-тестування - стадію виправлення помилок. При цьому, як правило, на кожному етапі розробки проміжні результати роботи доступні кінцевим користувачам.

•Реліз - випуск остаточної версії програми - готового для використання продукту. У релізі зазвичай збирають всі версії і оновлення, і випускають кінцевий продукт з усіма виправленнями, який не потрібно оновлювати, так як він є найновішою версією.

1.7. За стратегією тестування:

- *Тестування частин проти тестування цілого.* Існує висхідна і спадна стратегії тестування.

• *Висхідна стратегія* тестування полягає в наступному: спочатку тестуються окремі модулі, потім ці модулі інтегруються один з одним. Тестування спільної роботи модулів називається інтеграційним. Висхідне тестування дозволяє локалізувати помилки до першого модуля і якщо ви бачите баг при спільній роботі декількох оттестировать модулів, то це вже помилка інтеграції. Недолік висхідній стратегії тестування - написання спеціального коду оболонки для роботи окремого модуля, а також коду-заглушки, якщо цей модуль викликає наступний.

• *Низхідна стратегія* тестування виключає необхідність написання оболонки. Пишуть тільки заглушки, так як спочатку тестується верхній рівень ієрархії. Крім висхідній і низхідній стратегій тестування використовується тестування цілого продукту.

- Псевдоотладка і мутаційні тестування. Ці стратегії пов'язані з навмисним введенням помилок в код програми.

• псевдоотладка використовується для оцінки проведених тестів;

• мутаційне тестування , перш за все, використовується для перевірки якості проведених тестів. Впроваджується невелика зміна в програму, яка в результаті тестування обов'язково має проявитися і якщо цього не трапляється, то тести підібрані невдало.

1.8. За ознакою позитивності сценаріїв:

- Позитивне тестування (positive testing)
- Негативний тестування (negative testing)

1.9. За ступенем підготовленості до тестування:

- Тестування по документації (formal testing)
- Інтуїтивне тестування (adhoc testing)

Контрольні запитання

1. Охарактеризуйте класифікацію видів тестування п про об'єкту тестування.
2. Охарактеризуйте класифікацію видів тестування по повноті інформації про об'єкт тестування.
3. Охарактеризуйте класифікацію видів тестування за ступенем автоматизації процесу тестування.

4. Охарактеризуйте класифікацію видів тестування за ступенем ізольованості компонентів.
5. Охарактеризуйте класифікацію видів тестування за часом проведення тестування.
6. Охарактеризуйте класифікацію видів тестування за рівнем тестування.
7. Охарактеризуйте класифікацію видів тестування по стратегії тестування.
8. Охарактеризуйте класифікацію видів тестування за ознакою позитивності сценаріїв.
9. Охарактеризуйте класифікацію видів тестування за ступенем підготовленості до тестування.
10. Поясніть поняття «псевдоотладка і мутаційні тестування».