Міністерство освіти і науки України
Міжнародний гуманітарний університет
Кафедра інформаційних технологій

С.Б. Приходько

ЯКІСТЬ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Методичні вказівки до практичних робіт
для здобувачів вищої освіти,
які навчаються за спеціальностями
галузі «Інформаційні технології»

Одеса 2025

Затверджено Вченою Радою Міжнародного гуманітарного університету.
Протокол № 5 від 20 січня 2025 р.

С.Б. Приходько Якість та тестування програмного забезпечення. Методичні вказівки до практичних робіт для здобувачів вищої освіти, які навчаються за спеціальностями галузі «Інформаційні технології» [Електронне видання]. Кафедра інформаційних технологій Міжнародного гуманітарного університету. Одеса, 2025. 42 с.

Дисципліна «Якість та тестування програмного забезпечення» спрямована на формування у здобувачів вищої освіти знань і навичок забезпечення якості програмних систем на різних етапах їх життєвого циклу. Курс охоплює основні принципи забезпечення якості програмного забезпечення, моделі та характеристики якості програмних систем, процеси управління якістю, методи контролю якості та тестування програмного забезпечення. У межах дисципліни розглядаються рівні тестування програмного забезпечення, методи тестування типу «чорної скриньки» та «білої скриньки», метрики якості програмних систем, а також сучасні інструментальні засоби автоматизації тестування. Вивчення дисципліни сприяє формуванню здатності аналізувати якість програмного забезпечення, планувати та організовувати процес тестування, виявляти дефекти програмних систем та застосовувати сучасні інструменти контролю якості програмного забезпечення. Особлива увага приділяється інтеграції процесів тестування у життєвий цикл розроблення програмного забезпечення та використанню сучасних інструментальних засобів автоматизації тестування.

ЗМІСТ

ТЕМА 1. ОСНОВИ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Практична робота 1. Аналіз якості програмного забезпечення

Мета роботи – ознайомитися з поняттям якості програмного забезпечення, проаналізувати основні характеристики якості та атрибути програмних систем.

Ключові положення

Якість програмного забезпечення є комплексною характеристикою, що визначає здатність програмного продукту відповідати встановленим вимогам і очікуванням користувачів. Вона формується під впливом багатьох факторів, серед яких правильність реалізації функцій, стабільність роботи, ефективність використання ресурсів і зручність взаємодії з користувачем. Якість не є окремою властивістю, а є результатом узгодженої роботи всіх компонентів програмної системи.

У сучасній інженерії програмного забезпечення якість розглядається як сукупність характеристик, що можуть бути формалізовані та оцінені. Це дозволяє здійснювати об’єктивний аналіз програмних продуктів і порівнювати різні рішення. Для цього використовуються моделі якості, які визначають структуру характеристик і дають змогу систематизувати підхід до оцінювання.

До основних характеристик якості програмного забезпечення належать функціональна придатність, надійність, продуктивність, зручність використання, супроводжуваність та переносимість. Функціональна придатність визначає відповідність програмного забезпечення вимогам і правильність виконання функцій. Надійність характеризує стабільність роботи системи та її здатність працювати без відмов. Продуктивність відображає ефективність використання ресурсів і швидкість виконання операцій.

Зручність використання визначає, наскільки легко користувач може взаємодіяти з програмною системою. Супроводжуваність характеризує здатність програмного забезпечення до змін і виправлення помилок. Переносимість визначає можливість роботи системи у різних середовищах і на різних платформах. Кожна з цих характеристик має важливе значення для загальної якості програмного продукту.

Атрибути якості програмних систем деталізують характеристики і дозволяють більш точно оцінити окремі властивості. Наприклад, надійність може включати безвідмовність, відновлюваність і стійкість до помилок. Продуктивність може характеризуватися часом відгуку і пропускною здатністю. Такий підхід дозволяє виконувати більш глибокий аналіз якості.

Важливою особливістю є взаємозв’язок характеристик якості. Покращення однієї характеристики може впливати на інші. Наприклад, підвищення продуктивності може ускладнити структуру коду і знизити супроводжуваність. Тому під час розробки програмного забезпечення необхідно враховувати баланс між різними характеристиками.

Оцінювання якості програмного забезпечення базується на використанні метрик і процедур аналізу. Це дозволяє визначити рівень відповідності системи вимогам і виявити проблемні області. Результати такого аналізу використовуються для вдосконалення програмного продукту і підвищення його якості.

Таким чином, якість програмного забезпечення є багатогранною характеристикою, що визначає ефективність, надійність і придатність програмного продукту до використання. Розуміння її характеристик і атрибутів є необхідною умовою для розробки якісних програмних систем.

Практичне завдання

1. Обрати програмний продукт для аналізу (вебзастосунок, мобільний застосунок або десктопна програма).
2. Визначити основні характеристики якості обраного програмного забезпечення.
3. Проаналізувати функціональну придатність програмного продукту.
4. Оцінити надійність програмного забезпечення на основі спостережень або доступної інформації.
5. Проаналізувати продуктивність програмного продукту.
6. Оцінити зручність використання інтерфейсу.
7. Визначити рівень супроводжуваності програмного забезпечення (на основі відкритості коду або документації).
8. Проаналізувати переносимість програмного продукту.
9. Визначити взаємозв’язок між характеристиками якості.
10. Сформувати загальний висновок щодо рівня якості програмного забезпечення.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з поняттям якості програмного забезпечення, основними характеристиками та їх змістом. Особливу увагу слід приділити розумінню того, як різні характеристики впливають на загальну якість програмного продукту.

На першому етапі необхідно обрати програмний продукт для аналізу. Рекомендується використовувати програмне забезпечення, з яким студент має досвід роботи, оскільки це дозволить більш об’єктивно оцінити його характеристики. Це може бути вебзастосунок, мобільний застосунок або десктопна програма.

Далі необхідно визначити основні характеристики якості для обраного програмного забезпечення. Для цього слід проаналізувати функціональні можливості системи, її стабільність, швидкість роботи, зручність використання та інші властивості. Кожну характеристику необхідно описати окремо.

Під час аналізу функціональної придатності слід перевірити, чи відповідає програмний продукт заявленим можливостям. Необхідно визначити, чи всі функції працюють коректно і чи немає помилок у виконанні основних операцій.

Оцінювання надійності може виконуватися на основі спостережень за роботою системи. Слід звернути увагу на наявність збоїв, помилок і стабільність роботи. Для аналізу продуктивності необхідно оцінити швидкість виконання операцій і час відгуку системи.

Під час оцінювання зручності використання необхідно проаналізувати інтерфейс користувача, простоту виконання дій і зрозумілість елементів управління. Важливо визначити, наскільки легко користувач може виконувати типові завдання.

Для аналізу супроводжуваності слід звернути увагу на наявність документації, можливість внесення змін і зрозумілість структури програмного забезпечення. Якщо код є відкритим, можна оцінити його структуру та організацію.

Аналіз переносимості передбачає визначення можливості використання програмного продукту на різних пристроях і платформах. Необхідно перевірити, чи працює система у різних середовищах і чи не виникають проблеми сумісності.

На завершальному етапі необхідно проаналізувати взаємозв'язок характеристик якості та сформувати загальний висновок. Слід визначити сильні та слабкі сторони програмного продукту і оцінити його якість у цілому.

Контрольні питання

1. Що таке якість програмного забезпечення?
2. Які основні характеристики якості існують?
3. Що означає функціональна придатність?
4. Як визначається надійність програмного забезпечення?
5. Що характеризує продуктивність?
6. У чому полягає зручність використання?
7. Що означає супроводжуваність?
8. Як визначається переносимість?
9. Що таке атрибути якості?
10. Чому важливо враховувати взаємозв'язок характеристик якості?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис обраного програмного продукту;
– аналіз характеристик якості;
– оцінювання атрибутів якості;
– результати аналізу;
– висновки.

Практична робота 2. Оцінювання якості за моделлю ISO/IEC 25010

Мета роботи – ознайомитися з моделлю якості ISO/IEC 25010, проаналізувати характеристики та підхарактеристики якості програмного забезпечення і виконати оцінювання програмного продукту.

Ключові положення

Модель якості ISO/IEC 25010 є сучасним стандартом оцінювання якості програмного забезпечення, який визначає структурований підхід до аналізу властивостей програмного продукту. Вона є частиною серії стандартів SQuaRE і використовується для формування вимог до якості, її оцінювання та контролю. Модель дозволяє розглядати якість як

сукупність взаємопов’язаних характеристик, кожна з яких деталізується через підхарактеристики.

У межах моделі ISO/IEC 25010 виділяється якість продукту, яка охоплює внутрішні та зовнішні властивості програмного забезпечення. Вона включає такі характеристики, як функціональна придатність, продуктивність, сумісність, зручність використання, надійність, безпека, супроводжуваність та переносимість. Кожна з цих характеристик має чітке визначення і використовується для оцінювання різних властивостей програмного продукту.

Функціональна придатність визначає повноту, коректність і доцільність реалізації функцій. Вона відображає, наскільки програмне забезпечення виконує свої основні завдання. Продуктивність характеризує ефективність використання ресурсів і швидкість виконання операцій. Сумісність визначає здатність програмного забезпечення взаємодіяти з іншими системами. Зручність використання відображає простоту взаємодії користувача з програмною системою. Вона включає зрозумілість інтерфейсу, легкість навчання та ефективність виконання дій. Надійність визначає стабільність роботи системи і її здатність функціонувати без відмов.

Безпека характеризує захищеність даних і доступу до системи. Супроводжуваність визначає можливість внесення змін і виправлення помилок. Переносимість відображає здатність програмного забезпечення працювати у різних середовищах.

Кожна характеристика деталізується через підхарактеристики, що дозволяє виконувати більш точний аналіз. Наприклад, надійність може включати безвідмовність, відновлюваність і стійкість до помилок. Зручність використання може включати навчальність і керованість. Така деталізація дозволяє оцінювати якість більш об’єктивно.

Оцінювання якості за моделлю ISO/IEC 25010 передбачає визначення критеріїв для кожної характеристики, аналіз програмного продукту та формування висновків. Для цього можуть використовуватися як кількісні метрики, так і якісні оцінки. Результатом є комплексне уявлення про якість програмного забезпечення.

Важливою особливістю є те, що характеристики якості взаємопов’язані. Наприклад, підвищення рівня безпеки може впливати на продуктивність, а покращення функціональності може ускладнювати систему. Тому оцінювання повинно враховувати баланс між різними характеристиками.

Таким чином, модель ISO/IEC 25010 є універсальним інструментом для оцінювання якості програмного забезпечення, який дозволяє систематизувати аналіз і отримати обґрунтовані результати.

Практичне завдання

1. Обрати програмний продукт для оцінювання.
2. Ознайомитися зі структурою моделі ISO/IEC 25010.
3. Визначити основні характеристики якості для обраного продукту.
4. Для кожної характеристики визначити відповідні підхарактеристики.
5. Виконати оцінювання функціональної придатності.
6. Оцінити продуктивність програмного забезпечення.
7. Проаналізувати зручність використання.
8. Оцінити надійність програмного продукту.
9. Проаналізувати супроводжуваність і переносимість.
10. Сформувати загальний висновок щодо якості програмного забезпечення.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися зі структурою моделі ISO/IEC 25010 і зрозуміти зміст кожної характеристики якості. Особливу увагу слід приділити підхарактеристикам, оскільки саме вони дозволяють деталізувати оцінювання.

На першому етапі необхідно обрати програмний продукт для аналізу. Рекомендується використовувати програмне забезпечення, з яким є досвід роботи. Це дозволить виконати більш обґрунтоване оцінювання.

Далі необхідно визначити характеристики якості, які будуть аналізуватися. Для кожної характеристики потрібно виділити підхарактеристики і сформувати критерії оцінювання. Наприклад, для продуктивності можна використати час відгуку, для надійності – стабільність роботи.

Під час оцінювання необхідно використовувати як спостереження, так і доступну інформацію про програмний продукт. Для кожної характеристики слід надати опис і оцінку. Оцінювання може виконуватися у вигляді якісного опису або з використанням шкали.

Особливу увагу слід приділити обґрунтуванню оцінок. Кожне твердження повинно базуватися на конкретних фактах або прикладах використання програмного забезпечення. Це забезпечить об’єктивність результатів.

На завершальному етапі необхідно узагальнити результати оцінювання і сформувати висновок. Слід визначити сильні та слабкі сторони програмного продукту і оцінити його якість у цілому.

Контрольні питання

1. Що таке модель ISO/IEC 25010?
2. Які характеристики якості вона визначає?
3. Що таке підхарактеристики якості?
4. Що означає функціональна придатність?
5. Як визначається продуктивність?
6. Що характеризує зручність використання?
7. Які властивості визначають надійність?
8. Що означає супроводжуваність?
9. Як оцінюється переносимість?
10. Чому важливо враховувати взаємозв’язок характеристик?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис обраного програмного продукту;
– опис моделі ISO/IEC 25010;
– оцінювання характеристик і підхарактеристик;
– результати аналізу;
– висновки.

Практична робота 3. Метрики якості програмного забезпечення

Мета роботи – ознайомитися з метриками програмного забезпечення, навчитися застосовувати кількісні показники для оцінювання якості програмних систем.

Ключові положення

Метрики програмного забезпечення є числовими показниками, що використовуються для кількісного оцінювання властивостей програмного продукту або процесу його розробки. Вони дозволяють формалізувати оцінювання якості, забезпечити об’єктивність аналізу та виконувати порівняння між різними програмними системами або їх версіями.

Застосування метрик є важливим елементом інженерії програмного забезпечення, оскільки дозволяє переходити від суб’єктивних оцінок до кількісно обґрунтованих рішень. Метрики використовуються для контролю якості, виявлення проблемних областей і прийняття управлінських рішень.

Метрики програмного забезпечення поділяються на кілька категорій. Метрики продукту характеризують властивості програмного коду, такі як розмір, складність і структура. Метрики процесу відображають ефективність розробки, наприклад час виконання завдань або кількість змін. Метрики проєкту характеризують ресурси, строки та витрати.

Однією з базових метрик є кількість рядків коду. Вона використовується для оцінювання розміру програмного продукту, але не відображає його складність або якість. Тому для більш точного аналізу використовуються інші метрики, зокрема метрики складності.

Цикломатична складність визначає кількість незалежних шляхів виконання програми і використовується для оцінювання складності тестування. Чим вища складність, тим більше тестових випадків необхідно для перевірки програми. Це дозволяє оцінити ризик виникнення дефектів і складність супроводу.

Метрики дефектів є важливим показником якості програмного забезпечення. До них належать кількість дефектів, щільність дефектів та час їх усунення. Щільність дефектів визначається як відношення кількості дефектів до обсягу програмного коду і дозволяє оцінити надійність програмного продукту.

Метрики продуктивності характеризують ефективність роботи системи. До них належать час відгуку, пропускна здатність і використання ресурсів. Ці показники дозволяють оцінити, наскільки ефективно програмне забезпечення виконує свої функції.

Метрики покриття тестами визначають, яка частина програмного коду була перевірена під час тестування. Високий рівень покриття свідчить про більш повну перевірку, але не гарантує відсутності помилок. Тому ці метрики повинні використовуватися разом з іншими показниками.

Кількісні показники якості програмних систем дозволяють оцінити різні характеристики, такі як надійність, продуктивність, супроводжуваність і зручність використання. Вони можуть використовуватися для контролю якості на різних етапах життєвого циклу.

Важливою особливістю є те, що окремі метрики не дають повної картини якості. Тому необхідно використовувати їх у комплексі. Це дозволяє отримати більш об’єктивну оцінку і врахувати різні властивості програмного забезпечення.

Таким чином, метрики програмного забезпечення є важливим інструментом оцінювання якості, який дозволяє виконувати кількісний аналіз і приймати обґрунтовані рішення.

Практичне завдання

1. Обрати програмний продукт або програмний модуль для аналізу.
2. Визначити основні метрики, що будуть використовуватися для оцінювання.
3. Оцінити розмір програмного забезпечення.
4. Визначити складність програмного коду.
5. Проаналізувати кількість дефектів (за наявності даних).
6. Обчислити щільність дефектів.
7. Оцінити продуктивність програмного забезпечення.
8. Визначити рівень покриття тестами.
9. Проаналізувати отримані результати.
10. Сформувати висновок щодо якості програмного забезпечення.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з основними поняттями метрик програмного забезпечення та їх класифікацією. Особливу увагу слід приділити розумінню того, які метрики використовуються для оцінювання різних характеристик якості.

На першому етапі необхідно обрати програмний продукт або його частину для аналізу. Це може бути невеликий програмний модуль, відкритий проєкт або власна розробка. Важливо, щоб були доступні дані для оцінювання.

Далі необхідно визначити перелік метрик, які будуть використовуватися. Рекомендується обрати метрики, що характеризують різні властивості програмного забезпечення, зокрема розмір, складність, надійність і продуктивність.

Для оцінювання розміру програмного забезпечення слід визначити кількість рядків коду. Для аналізу складності можна використати оцінку кількості умов і циклів у програмі. За можливості доцільно використати інструменти статичного аналізу.

Для аналізу дефектів необхідно зібрати інформацію про кількість помилок. Якщо така інформація недоступна, можна виконати умовне оцінювання. Щільність дефектів визначається як відношення кількості дефектів до обсягу коду.

Оцінювання продуктивності може виконуватися шляхом вимірювання часу виконання основних операцій. Для цього можна використати прості експерименти або засоби вимірювання.

Рівень покриття тестами визначається на основі аналізу виконаних тестів. За відсутності автоматизованих засобів можна виконати оцінювання вручну.

Після отримання всіх показників необхідно виконати їх аналіз. Слід визначити сильні та слабкі сторони програмного забезпечення і оцінити його якість.

Контрольні питання

1. Що таке метрики програмного забезпечення?
2. Які основні види метрик існують?

3. Що характеризують метрики продукту?
4. Що таке цикломатична складність?
5. Як визначається щільність дефектів?
6. Які метрики використовуються для оцінювання продуктивності?
7. Що таке покриття тестами?
8. Чому важливо використовувати кілька метрик?
9. Які обмеження мають метрики?
10. Як метрики використовуються для оцінювання якості?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис обраного програмного продукту;
– перелік використаних метрик;
– результати обчислень;
– аналіз показників;
– висновки.

ТЕМА 2. ПРОЦЕСИ ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Практична робота 4. План забезпечення якості програмного забезпечення

Мета роботи – ознайомитися з процесами забезпечення якості програмного забезпечення та розробити план забезпечення якості для програмного проєкту.

Ключові положення

Забезпечення якості програмного забезпечення є організованим процесом, спрямованим на попередження дефектів і забезпечення відповідності програмного продукту встановленим вимогам. Воно охоплює всі етапи життєвого циклу і включає планування, виконання перевірок, аналіз результатів та вдосконалення процесів розробки.

Процеси забезпечення якості визначають структуру діяльності, яка дозволяє контролювати якість програмного продукту на всіх етапах створення. До таких процесів належать аналіз вимог, рецензування, тестування, аудит і контроль змін. Вони повинні бути узгоджені між собою і інтегровані у загальний процес розробки.

Планування забезпечення якості є ключовим етапом, на якому визначаються цілі якості, методи їх досягнення та критерії оцінювання. План забезпечення якості є документом, що описує організацію робіт, відповідальність учасників, використовувані методи і засоби, а також порядок контролю результатів.

У плані забезпечення якості визначаються стандарти і правила, яких необхідно дотримуватися під час розробки програмного забезпечення. Це можуть бути стандарти кодування, правила документування, вимоги до тестування та інші регламенти. Дотримання цих стандартів дозволяє забезпечити узгодженість і підвищити якість програмного продукту.

Важливим елементом є визначення метрик якості, які використовуються для оцінювання результатів. Це можуть бути показники кількості дефектів, покриття тестами, часу виконання операцій та інші. Метрики дозволяють виконувати об’єктивний контроль якості і оцінювати ефективність процесів.

План забезпечення якості також включає опис процедур верифікації та валідації. Верифікація забезпечує перевірку відповідності продукту технічним вимогам, а валідація – відповідність потребам користувача. Ці процедури виконуються на різних етапах розробки і забезпечують комплексний підхід до контролю якості.

Особливу увагу слід приділити управлінню ризиками, пов’язаними з якістю. У процесі планування необхідно визначити можливі ризики, оцінити їх вплив і розробити заходи для їх мінімізації. Це дозволяє зменшити ймовірність виникнення проблем і підвищити стабільність процесу розробки.

Таким чином, план забезпечення якості є основним документом, що визначає організацію робіт з якості у програмному проєкті. Його розробка дозволяє систематизувати процеси і забезпечити досягнення встановлених вимог.

Практичне завдання

1. Обрати програмний проєкт для розробки плану забезпечення якості.
2. Визначити основні процеси забезпечення якості у проєкті.

3. Сформулювати цілі забезпечення якості.
4. Визначити стандарти і правила, що будуть застосовуватися.
5. Визначити метрики якості для оцінювання.
6. Розробити процедури верифікації.
7. Розробити процедури валідації.
8. Визначити ризики, пов’язані з якістю.
9. Розробити заходи для мінімізації ризиків.
10. Сформувати план забезпечення якості.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з процесами забезпечення якості програмного забезпечення та їх роллю у розробці програмних систем. Особливу увагу слід приділити розумінню структури плану забезпечення якості.

На першому етапі необхідно обрати програмний проєкт. Це може бути навчальний проєкт, реальний програмний продукт або умовна система. Важливо визначити його призначення, основні функції і особливості.

Далі необхідно визначити процеси забезпечення якості, які будуть використовуватися у проєкті. Слід описати, які перевірки будуть виконуватися на різних етапах життєвого циклу.

Після цього необхідно сформулювати цілі забезпечення якості. Вони повинні відображати вимоги до програмного продукту і визначати очікуваний рівень якості.

Наступним етапом є визначення стандартів і правил. Необхідно описати, які стандарти будуть використовуватися у процесі розробки і як буде контролюватися їх дотримання.

Далі необхідно визначити метрики якості. Для кожної метрики слід описати спосіб її обчислення і використання. Це дозволить забезпечити об’єктивність оцінювання.

Після цього необхідно розробити процедури верифікації і валідації. Слід визначити, які методи будуть використовуватися для перевірки програмного забезпечення.

Важливим етапом є аналіз ризиків. Необхідно визначити можливі проблеми, що можуть вплинути на якість, і розробити заходи для їх усунення або зменшення їх впливу.

На завершальному етапі необхідно оформити план забезпечення якості у вигляді документа. Він повинен містити всі визначені елементи і бути логічно структурованим.

Контрольні питання

1. Що таке забезпечення якості програмного забезпечення?
2. Які процеси входять до забезпечення якості?
3. Що таке план забезпечення якості?
4. Які елементи містить план?
5. Що таке верифікація?
6. Що таке валідація?
7. Яку роль відіграють метрики якості?
8. Що таке ризики якості?
9. Як виконується планування якості?
10. Чому важливо документувати процеси забезпечення якості?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис програмного проєкту;
– опис процесів забезпечення якості;
– план забезпечення якості;
– аналіз ризиків;
– висновки..

Практична робота 5. Управління конфігураціями та версіями

Мета роботи – ознайомитися з процесами управління конфігураціями та змінами програмного забезпечення, дослідити застосування систем контролю версій у процесі розробки.

Ключові положення

Управління конфігураціями програмного забезпечення є процесом контролю складу програмного продукту, його стану та змін протягом життєвого циклу. Воно забезпечує цілісність програмної системи, узгодженість її компонентів і можливість відтворення будь-якої версії продукту. У складних проєктах цей процес є критично важливим, оскільки дозволяє уникнути помилок, пов'язаних із неконтрольованими змінами.

Конфігурація програмного забезпечення містить усі елементи системи, включаючи вихідний код, виконувані файли, бібліотеки, документацію та налаштування. Кожен із цих елементів повинен бути ідентифікований і підлягати контролю. Це дозволяє відслідковувати зміни і забезпечувати узгодженість між різними частинами системи.

Процес управління конфігураціями включає ідентифікацію конфігураційних елементів, контроль змін, облік стану та аудит. Ідентифікація дозволяє визначити структуру системи. Контроль змін забезпечує впорядковане внесення змін. Облік стану дозволяє відслідковувати поточний стан системи. Аудит забезпечує перевірку відповідності фактичного стану задокументованому.

Управління змінами є складовою управління конфігураціями і визначає порядок внесення змін у програмний продукт. Зміни можуть бути пов'язані з виправленням помилок, додаванням нових функцій або оптимізацією. Важливою умовою є формалізація процесу змін, що включає їх реєстрацію, аналіз і затвердження.

Кожна зміна повинна бути задокументована і проходити певні етапи обробки. Це дозволяє забезпечити прозорість процесу і уникнути хаотичних змін. Важливим є також оцінювання впливу змін на інші компоненти системи.

Системи контролю версій є основним інструментом реалізації управління конфігураціями. Вони дозволяють зберігати історію змін, відслідковувати модифікації та координувати роботу команди. Використання таких систем забезпечує можливість повернення до попередніх версій і аналізу змін.

Сучасні системи контролю версій підтримують гілки розробки, що дозволяє паралельно виконувати різні завдання. Це забезпечує гнучкість і підвищує ефективність роботи команди. Також вони дозволяють об’єднувати зміни і вирішувати конфлікти.

У практиці широко використовується Git як розподілена система контролю версій, що забезпечує високу гнучкість і надійність. Також застосовуються Subversion та Mercurial. Вибір системи залежить від вимог проєкту і особливостей організації роботи.

Використання систем контролю версій дозволяє реалізувати ефективні підходи до організації розробки, зокрема використання гілок для різних функцій і версій продукту. Це дозволяє зменшити ризики і забезпечити стабільність програмного забезпечення.

Таким чином, управління конфігураціями та змінами програмного забезпечення є важливим процесом, що забезпечує контроль за розвитком програмного продукту. Використання систем контролю версій дозволяє ефективно реалізувати цей процес і підвищити якість програмного забезпечення.

Практичне завдання

1. Обрати програмний проєкт або створити новий.
2. Визначити конфігураційні елементи програмного забезпечення.
3. Описати структуру конфігурації.
4. Проаналізувати процес управління змінами.
5. Встановити систему контролю версій.
6. Ініціалізувати репозиторій для проєкту.
7. Виконати додавання і фіксацію змін.
8. Створити гілку для окремої функції.
9. Виконати об’єднання гілок.
10. Проаналізувати історію змін.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з поняттями конфігурації програмного забезпечення та управління змінами. Особливу увагу слід приділити ролі систем контролю версій у процесі розробки.

На першому етапі необхідно обрати програмний проєкт або створити новий. Рекомендується використовувати простий проєкт, що дозволить зосередитися на процесах управління конфігураціями.

Далі необхідно визначити конфігураційні елементи. Слід виділити всі компоненти, що входять до складу програмного продукту, і описати їх.

Після цього необхідно проаналізувати процес управління змінами. Слід визначити, як будуть фіксуватися зміни, як буде виконуватися їх аналіз і затвердження.

Наступним етапом є встановлення системи контролю версій. Для цього можна використати Git. Після встановлення необхідно ініціалізувати репозиторій і додати файли проєкту.

Далі необхідно виконати фіксацію змін. Слід внести зміни у файли і зафіксувати їх у репозиторії. Це дозволить сформувати історію змін.

Після цього необхідно створити гілку для окремої функції. У цій гілці слід внести зміни і зафіксувати їх. Потім необхідно виконати об’єднання гілок.

На завершальному етапі необхідно проаналізувати історію змін. Слід звернути увагу на структуру комітів і процес розвитку проєкту.

Контрольні питання

1. Що таке конфігурація програмного забезпечення?
2. Які елементи входять до конфігурації?
3. Що таке управління змінами?
4. Які етапи обробки змін існують?
5. Що таке система контролю версій?
6. Які функції виконують системи контролю версій?
7. Що таке репозиторій?
8. Що таке гілка?
9. Як виконується об’єднання змін?
10. Чому важливо відслідковувати історію змін?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис проєкту;
– опис конфігураційних елементів;
– результати роботи з системою контролю версій;
– аналіз змін;
– висновки.

ТЕМА 3. ОСНОВИ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Практична робота 6. Аналіз процесу тестування програмного забезпечення

Мета роботи – ознайомитися з процесом тестування програмного забезпечення, проаналізувати його основні етапи, а також розглянути поняття верифікації та валідації.

Ключові положення

Тестування програмного забезпечення є систематичним процесом перевірки програмного продукту з метою виявлення дефектів, оцінювання його якості та підтвердження відповідності встановленим вимогам. Воно є невід'ємною складовою інженерії програмного забезпечення і виконується протягом усього життєвого циклу. Основне завдання тестування полягає не лише у виявленні помилок, а й у забезпеченні впевненості у правильності роботи системи.

Процес тестування включає низку взаємопов'язаних дій, серед яких аналіз вимог, планування тестування, розробка тестових випадків, виконання тестів і аналіз результатів. Кожен із цих етапів має власну роль у забезпеченні якості програмного продукту. Важливою особливістю є те, що тестування повинно бути організоване як безперервний процес, а не як окремий завершальний етап.

Поняття тестування тісно пов'язане з перевіркою правильності реалізації функцій і відповідності системи вимогам. При цьому важливо розрізняти виявлення дефектів і підтвердження якості. Тестування дозволяє виявити помилки, але не гарантує їх повну відсутність. Тому результати тестування повинні інтерпретуватися з урахуванням обмежень цього процесу.

Верифікація є процесом перевірки відповідності програмного забезпечення технічним специфікаціям. Вона відповідає на питання, чи правильно реалізовано систему відповідно до заданих вимог. Верифікація може включати аналіз документації, рецензування коду, статичний аналіз і виконання тестів. Вона орієнтована на внутрішню правильність реалізації.

Валідація, у свою чергу, визначає відповідність програмного забезпечення потребам користувачів. Вона відповідає на питання, чи створено правильний продукт. Валідація виконується у реальних або наближених до реальних умовах і дозволяє оцінити, наскільки система задовольняє очікування користувачів.

Важливою є взаємодія верифікації та валідації. Верифікація забезпечує правильність реалізації, тоді як валідація підтверджує доцільність створеного продукту. Відсутність одного з цих процесів може призвести до створення або технічно правильного, але непотрібного продукту, або корисного, але реалізованого з помилками.

У процесі тестування також важливим є визначення критеріїв якості та умов завершення тестування. Це дозволяє оцінити, чи досягнуто необхідного рівня якості і чи готовий продукт до впровадження. Без чітких критеріїв тестування може бути нескінченним або недостатнім.

Таким чином, тестування програмного забезпечення є комплексним процесом, що забезпечує контроль якості на всіх етапах розробки. Верифікація і валідація доповнюють один одного і дозволяють оцінити як технічну правильність, так і практичну цінність програмного продукту.

Практичне завдання

1. Обрати програмний продукт для аналізу.
2. Визначити основні етапи процесу тестування.
3. Проаналізувати процес планування тестування.
4. Визначити методи тестування, що використовуються.
5. Проаналізувати процес виконання тестів.
6. Оцінити процес аналізу результатів тестування.
7. Визначити приклади верифікації у проєкті.
8. Визначити приклади валідації у проєкті.
9. Порівняти верифікацію та валідацію.
10. Сформувати висновок щодо ефективності процесу тестування.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з поняттям тестування програмного забезпечення та його роллю у забезпеченні якості. Особливу увагу слід приділити відмінностям між верифікацією та валідацією, а також їх місцю у процесі розробки.

На першому етапі необхідно обрати програмний продукт для аналізу. Рекомендується використовувати програмне забезпечення, з яким є практичний досвід роботи. Це дозволить виконати більш об’єктивний аналіз процесу тестування.

Далі необхідно визначити етапи тестування, які застосовуються у проєкті. Слід описати, як виконується аналіз вимог, планування тестування, підготовка тестових випадків і виконання тестів. Необхідно звернути увагу на послідовність цих етапів і їх взаємозв’язок.

Під час аналізу процесу тестування необхідно визначити, які методи тестування використовуються. Це можуть бути як ручні, так і автоматизовані підходи. Важливо оцінити їх ефективність і доцільність використання.

Особливу увагу слід приділити процесу виконання тестів і аналізу результатів. Необхідно визначити, як фіксуються результати, як обробляються дефекти і як приймаються рішення щодо якості програмного продукту.

Для аналізу верифікації необхідно визначити приклади перевірки відповідності технічним вимогам. Це можуть бути рецензування коду, перевірка документації або виконання тестів. Для аналізу валідації необхідно визначити, як перевіряється відповідність потребам користувачів.

На завершальному етапі необхідно виконати порівняння верифікації та валідації. Слід визначити їх роль у забезпеченні якості і зробити висновки щодо їх ефективності.

Рекомендується також проаналізувати можливі проблеми у процесі тестування, такі як недостатнє покриття тестами або відсутність чітких критеріїв оцінювання. Це дозволить більш глибоко зрозуміти особливості процесу тестування.

Контрольні питання

1. Що таке тестування програмного забезпечення?
2. Які основні етапи процесу тестування існують?

3. У чому полягає мета тестування?
4. Що таке верифікація?
5. Що таке валідація?
6. У чому різниця між верифікацією та валідацією?
7. Які методи тестування існують?
8. Як аналізуються результати тестування?
9. Які проблеми можуть виникати у процесі тестування?
10. Чому тестування є важливим етапом розробки?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис програмного продукту;
– аналіз процесу тестування;
– аналіз верифікації та валідації;
– результати дослідження;
– висновки.

Практична робота 7. Рівні тестування програмного забезпечення

Мета роботи – дослідити рівні тестування програмного забезпечення, проаналізувати особливості модульного, інтеграційного, системного та приймального тестування.

Ключові положення

Рівні тестування програмного забезпечення визначають етапи перевірки системи відповідно до ступеня її інтеграції. Такий підхід дозволяє послідовно перевіряти програмний продукт, починаючи з окремих компонентів і завершуючи оцінюванням системи в цілому. Використання рівнів тестування забезпечує структурованість процесу перевірки та підвищує ефективність виявлення дефектів.

Модульне тестування є початковим рівнем, на якому перевіряються окремі компоненти програмного забезпечення, такі як функції, методи або класи. Основною метою є перевірка правильності реалізації логіки кожного модуля. Модульне тестування виконується ізольовано від інших частин системи, що дозволяє точно визначити джерело помилки. Для цього можуть використовуватися спеціальні допоміжні засоби, які імітують взаємодію з іншими компонентами.

Інтеграційне тестування виконується після модульного і спрямоване на перевірку взаємодії між компонентами системи. На цьому рівні перевіряється правильність обміну даними, узгодженість інтерфейсів і коректність інтеграції модулів. Інтеграційне тестування дозволяє виявити дефекти, які не проявляються при ізольованому тестуванні окремих компонентів.

Системне тестування передбачає перевірку програмної системи як єдиного цілого. Воно виконується у середовищі, наближеному до реальних умов експлуатації, і дозволяє

оцінити відповідність системи функціональним і нефункціональним вимогам. На цьому рівні перевіряється поведінка системи при різних сценаріях використання.

Приймальне тестування є завершальним рівнем і виконується з метою підтвердження готовності програмного продукту до використання. Воно проводиться замовником або кінцевими користувачами і базується на вимогах до системи. Основною метою є перевірка того, чи відповідає програмне забезпечення очікуванням користувача.

Кожен рівень тестування має власну роль у забезпеченні якості програмного забезпечення. Модульне тестування дозволяє виявити помилки на ранніх етапах, інтеграційне забезпечує правильність взаємодії компонентів, системне дозволяє оцінити роботу системи в цілому, а приймальне підтверджує її готовність до використання.

Важливою особливістю є взаємозв'язок між рівнями тестування. Недоліки, не виявлені на ранніх етапах, можуть ускладнювати тестування на наступних рівнях. Тому ефективність тестування значною мірою залежить від повноти виконання перевірок на кожному рівні.

Таким чином, використання рівнів тестування дозволяє організувати процес перевірки програмного забезпечення і забезпечити поетапне виявлення дефектів.

Практичне завдання

1. Обрати програмний продукт або програмний модуль для дослідження.
2. Визначити компоненти, що можуть бути протестовані на модульному рівні.
3. Розробити приклади модульних тестів.
4. Визначити взаємозв'язки між компонентами системи.
5. Проаналізувати інтеграційне тестування.
6. Визначити сценарії системного тестування.
7. Описати умови приймального тестування.
8. Порівняти рівні тестування між собою.
9. Проаналізувати ефективність кожного рівня.
10. Сформувати загальний висновок.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з поняттям рівнів тестування та їх роллю у процесі забезпечення якості. Особливу увагу слід приділити розумінню відмінностей між модульним, інтеграційним, системним і приймальним тестуванням.

На першому етапі необхідно обрати програмний продукт або його частину. Рекомендується використовувати невеликий проєкт або модуль, що дозволить детально проаналізувати всі рівні тестування.

Далі необхідно визначити компоненти системи, які можуть бути протестовані на модульному рівні. Слід виділити окремі функції або класи і розробити для них тестові випадки. Необхідно перевірити правильність виконання основних операцій.

Після цього необхідно проаналізувати взаємодію між компонентами системи. Слід визначити, які частини системи взаємодіють між собою і як це впливає на тестування. На основі цього потрібно описати інтеграційне тестування.

На наступному етапі необхідно визначити сценарії системного тестування. Слід описати типові дії користувача і перевірити, як система реагує на різні ситуації. Важливо врахувати як стандартні, так і нестандартні сценарії.

Для аналізу приймального тестування необхідно визначити критерії, за якими буде оцінюватися готовність програмного продукту. Слід описати, які вимоги повинні бути виконані для прийняття системи.

Після виконання всіх етапів необхідно порівняти рівні тестування між собою. Слід визначити їх роль і ефективність, а також зробити висновки щодо їх використання.

Рекомендується також звернути увагу на взаємозв'язок між рівнями тестування і проаналізувати, як результати одного рівня впливають на інші. Це дозволить краще зрозуміти структуру процесу тестування.

Контрольні питання

1. Що таке рівні тестування програмного забезпечення?
2. У чому полягає модульне тестування?
3. Яка мета інтеграційного тестування?
4. Що перевіряється на системному рівні?
5. У чому полягає приймальне тестування?
6. Яка різниця між рівнями тестування?
7. Чому важливо виконувати тестування на всіх рівнях?
8. Які дефекти виявляються на різних рівнях?
9. Як рівні тестування взаємодіють між собою?
10. Яке значення має приймальне тестування?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис програмного продукту;
– аналіз модульного тестування;
– аналіз інтеграційного тестування;
– аналіз системного тестування;
– аналіз приймального тестування;
– висновки.

Практична робота 8. Тестові сценарії та Git

Мета роботи – навчитися розробляти тестові сценарії та тестові випадки, використовувати систему контролю версій Git і платформу GitHub для організації тестування, а також виконувати документування результатів.

Ключові положення

Розробка тестових сценаріїв і тестових випадків є основою процесу тестування програмного забезпечення. Тестовий сценарій описує загальну логіку перевірки певної функціональності, тоді як тестовий випадок містить конкретні дії, вхідні дані та очікувані результати. Якість тестових матеріалів безпосередньо впливає на ефективність тестування та повноту перевірки програмного продукту.

Планування тестування передбачає визначення обсягу тестових робіт, вибір підходів до перевірки, формування тестових сценаріїв і визначення критеріїв оцінювання. На цьому етапі важливо забезпечити узгодженість між вимогами та тестами. Це дозволяє гарантувати, що всі функціональні можливості програмного забезпечення будуть перевірені.

Тестові випадки повинні бути чітко сформульовані, однозначні та відтворювані. Вони повинні містити опис передумов, послідовність дій, вхідні дані та очікувані результати. Особлива увага приділяється перевірці граничних умов і нестандартних сценаріїв, що дозволяє виявити дефекти, які не проявляються при звичайному використанні системи.

Використання систем контролю версій, таких як Git, дозволяє організувати зберігання тестових матеріалів, відслідковувати їх зміни та забезпечувати узгодженість роботи команди. Кожна зміна тестових сценаріїв або результатів тестування може бути зафіксована у репозиторії, що забезпечує прозорість процесу.

Платформа GitHub дозволяє організувати спільну роботу над тестовими матеріалами, зберігати документацію, а також використовувати інструменти для відстеження змін і обговорення результатів. Це забезпечує ефективну взаємодію між учасниками проєкту.

Документування результатів тестування є важливим елементом процесу. Воно передбачає фіксацію виконаних тестів, отриманих результатів і виявлених дефектів. Документація дозволяє аналізувати якість програмного забезпечення, відслідковувати зміни та приймати обґрунтовані рішення.

Важливою є також трасованість між вимогами і тестами. Кожен тестовий випадок повинен бути пов’язаний із відповідною вимогою, що дозволяє контролювати повноту перевірки. Це забезпечує узгодженість процесу тестування і підвищує його ефективність.

Таким чином, розробка тестових сценаріїв, використання систем контролю версій і документування результатів є взаємопов’язаними елементами процесу тестування, що забезпечують його ефективність і прозорість.

Практичне завдання

1. Обрати програмний проєкт для тестування.
2. Визначити вимоги до програмного забезпечення.
3. Розробити тестові сценарії.
4. Сформувати тестові випадки.
5. Встановити Git.

6. Створити репозиторій і додати тестову документацію.
7. Виконати фіксацію змін тестових матеріалів.
8. Опублікувати проєкт на GitHub.
9. Виконати тестування і зафіксувати результати.
10. Оформити звіт про тестування.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з поняттями тестових сценаріїв і тестових випадків, а також з основами планування тестування. Особливу увагу слід приділити структурі тестових випадків і принципам їх формування.

На першому етапі необхідно обрати програмний проєкт. Це може бути невеликий застосунок або окремий модуль. Важливо визначити основні вимоги до системи, які будуть використовуватися для формування тестів.

Далі необхідно розробити тестові сценарії. Слід описати основні варіанти використання системи та визначити дії, які будуть перевірятися. На основі сценаріїв необхідно сформувати тестові випадки, що містять детальний опис перевірок.

Наступним етапом є організація роботи з системою контролю версій. Необхідно встановити Git, створити репозиторій і додати до нього тестові матеріали. Після цього слід виконати фіксацію змін і перевірити історію комітів.

Далі необхідно створити віддалений репозиторій на платформі GitHub і опублікувати проєкт. Це дозволить забезпечити зберігання тестових матеріалів і організувати спільну роботу.

Після цього необхідно виконати тестування відповідно до розроблених тестових випадків. Результати тестування слід зафіксувати у вигляді таблиці або текстового опису. У разі виявлення дефектів необхідно описати їх.

На завершальному етапі необхідно оформити документацію тестування. Слід описати виконані перевірки, отримані результати та зробити висновки щодо якості програмного забезпечення.

Рекомендується також проаналізувати, як використання систем контролю версій впливає на організацію тестування. Це дозволить оцінити ефективність застосування таких інструментів.

Контрольні питання

1. Що таке тестовий сценарій?
2. Що таке тестовий випадок?
3. Яка структура тестового випадку?
4. Що таке планування тестування?
5. Яку роль відіграє Git у тестуванні?
6. Для чого використовується GitHub?
7. Як виконується документування результатів тестування?
8. Що таке трасованість тестів?
9. Як фіксуються зміни у тестових матеріалах?
10. Чому важливо зберігати історію змін?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис програмного проєкту;
– тестові сценарії;
– тестові випадки;
– результати тестування;
– опис використання Git та GitHub;
– висновки.

ТЕМА 4. МЕТОДИ ТА ТЕХНІКИ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Практична робота 9. Тестові випадки за методом «чорної скриньки»

Мета роботи – навчитися розробляти тестові випадки із використанням методів тестування типу чорної скриньки, застосовувати метод еквівалентних класів і метод граничних значень для перевірки програмного забезпечення.

Ключові положення

Тестування типу чорної скриньки є підходом до перевірки програмного забезпечення, за якого внутрішня структура програми, програмний код і спосіб реалізації алгоритмів не аналізуються. Увага зосереджується на вхідних даних, виконуваних діях і результатах роботи системи. Такий підхід дає змогу перевірити відповідність програмного забезпечення встановленим вимогам з позиції користувача або замовника. У цьому випадку важливим є не те, як саме реалізована функція, а те, чи дає вона правильний результат для заданих умов.

Методи чорної скриньки широко застосовуються під час функціонального тестування, системного тестування та приймального тестування. Вони є зручними для перевірки інтерфейсів, форм введення даних, логіки обробки запитів, механізмів авторизації, звітів, обчислень та інших функціональних можливостей системи. Оскільки такі методи не потребують доступу до програмного коду, вони можуть використовуватися не лише розробниками, а й тестувальниками, аналітиками та іншими учасниками проєкту.

Одним із найпоширеніших методів тестування типу чорної скриньки є метод еквівалентних класів. Його суть полягає у поділі множини вхідних даних на окремі групи, для яких очікується однакова поведінка системи. Кожна така група називається класом еквівалентності. Якщо припускається, що система обробляє всі значення з одного класу однаково, то для перевірки достатньо вибрати один або кілька представників цього класу, а не тестувати всі можливі значення. Це дозволяє суттєво зменшити кількість тестових випадків без значної втрати якості перевірки.

Еквівалентні класи можуть бути допустимими та недопустимими. Допустимі класи містять значення, які відповідають вимогам до вхідних даних і повинні бути коректно оброблені системою. Недопустимі класи містять дані, які порушують встановлені обмеження, і використовуються для перевірки механізмів виявлення помилок та реакції системи на некоректне введення. Наприклад, якщо поле допускає введення цілого числа від 1 до 100, то значення 50 належить до допустимого класу, а значення 0, 101 або текстовий рядок – до недопустимих класів.

Метод еквівалентних класів є ефективним тоді, коли система має чітко визначені діапазони значень, обмеження на формат введення, перелік допустимих категорій або інші формалізовані правила. Він дозволяє структурувати тестування і зосередитися на дійсно важливих наборах даних. Разом із тим цей метод вимагає уважного аналізу вимог, оскільки помилка у визначенні класів може призвести до пропуску важливих перевірок.

Іншим важливим методом є метод граничних значень. Він базується на практичному спостереженні, що помилки часто виникають поблизу меж допустимих інтервалів. Якщо вимога визначає нижню та верхню межі допустимих значень, то саме ці межі та найближчі

до них значення мають бути обов’язково перевірені. Наприклад, якщо допустимий діапазон становить від 1 до 100, доцільно протестувати значення 1, 100, а також значення 0 і 101. У багатьох випадках також перевіряються сусідні допустимі значення, наприклад 2 і 99.

Метод граничних значень доповнює метод еквівалентних класів. Якщо перший дозволяє скоротити кількість тестів за рахунок узагальнення вхідних даних, то другий спрямований на більш детальну перевірку критичних точок, у яких найчастіше виникають дефекти. До таких дефектів належать помилки у логічних порівняннях, неправильне використання операторів більше або дорівнює, менше або дорівнює, а також некоректна перевірка мінімально або максимально допустимих значень.

Під час розробки тестових випадків важливо правильно формулювати очікуваний результат. Тестовий випадок не повинен обмежуватися лише описом введених даних. Він повинен містити умови виконання, послідовність дій, вхідні значення, очікуваний результат і, за потреби, примітки щодо середовища тестування. Лише за наявності чітко визначеного очікуваного результату можна зробити обґрунтований висновок про успішність або неуспішність перевірки.

Якість тестових випадків безпосередньо впливає на ефективність тестування. Нечітко сформульований тестовий випадок може бути виконаний неправильно або неоднозначно інтерпретований. Тому тестові випадки повинні бути зрозумілими, лаконічними, відтворюваними та пов’язаними з конкретними вимогами. Бажано також забезпечити трасування між вимогами та тестами, щоб було зрозуміло, яка саме вимога перевіряється кожним тестом.

У реальних проєктах методи чорної скриньки часто застосовуються одночасно. Наприклад, для форми реєстрації користувача можна спочатку визначити класи еквівалентності для поля віку, електронної адреси або пароля, а потім окремо перевірити граничні значення для довжини пароля, віку або кількості символів у полі вводу. Такий підхід дає змогу отримати більш повну перевірку при раціональній кількості тестів.

Таким чином, тестування типу чорної скриньки є важливим методом перевірки програмного забезпечення, орієнтованим на функціональні вимоги та поведінку системи. Метод еквівалентних класів дозволяє скоротити кількість тестів шляхом групування вхідних даних, а метод граничних значень забезпечує детальнішу перевірку критичних точок. Їх правильне застосування є основою якісного проєктування тестових випадків.

Практичне завдання

1. Обрати програмний продукт, вебформу або окремий програмний модуль для тестування.
2. Визначити функцію або набір функцій, для яких будуть розроблятися тестові випадки.
3. Проаналізувати вимоги до вхідних даних.
4. Виділити допустимі та недопустимі еквівалентні класи.
5. Визначити граничні значення для кожного з виділених класів.
6. Розробити тестові випадки на основі еквівалентних класів.
7. Розробити тестові випадки на основі граничних значень.
8. Для кожного тестового випадку визначити очікуваний результат.
9. Виконати тестування або змоделювати виконання тестових випадків.
10. Проаналізувати результати та сформувати висновок щодо повноти перевірки.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з поняттям тестування типу чорної скриньки та особливостями його застосування. Потрібно чітко усвідомити, що у цій роботі увага приділяється не програмному коду, а поведінці системи для різних наборів вхідних даних. Особливу увагу слід приділити аналізу вимог, оскільки саме вони є основою для побудови тестових випадків.

На першому етапі необхідно обрати об’єкт тестування. Це може бути форма введення даних, модуль авторизації, калькулятор, функція перевірки пароля, механізм реєстрації користувача або інший елемент програмного забезпечення, для якого можна чітко визначити вхідні дані та очікувані результати. Бажано обирати приклад із простими і зрозумілими правилами обробки даних, щоб основну увагу можна було зосередити на методиці розробки тестів.

Далі необхідно детально проаналізувати вимоги до обраної функції. Потрібно визначити, які значення є допустимими, які обмеження встановлено на формат введення, які мінімальні та максимальні межі існують, а також які помилки повинна виявляти система. Якщо у вимогах є нечіткі або суперечливі формулювання, їх слід окремо зафіксувати, оскільки це є важливим результатом аналізу тестованості.

Після цього необхідно виділити еквівалентні класи. Для кожного вхідного параметра слід визначити хоча б один допустимий клас і один або кілька недопустимих класів. Наприклад, для числового поля окремо можуть бути виділені допустимі значення у межах діапазону, значення нижче мінімальної межі, значення вище максимальної межі, порожнє поле, текстові символи замість числа. Для текстових полів можуть бути виділені класи за довжиною рядка, наявністю спеціальних символів, порожнім значенням або неправильним форматом.

На наступному етапі слід визначити граничні значення. Для кожного діапазону необхідно виписати нижню межу, верхню межу, значення безпосередньо перед межею та після неї. Якщо поле має обмеження за довжиною, потрібно перевірити рядки мінімальної допустимої довжини, максимальної допустимої довжини, коротші та довші за допустимі. При цьому слід не лише перелічити значення, а й обґрунтувати, чому саме вони є важливими для перевірки.

Після визначення класів і граничних значень необхідно оформити тестові випадки. Кожний тестовий випадок повинен містити номер, назву або короткий опис, передумови виконання, тестові дані, послідовність дій і очікуваний результат. Якщо використовується таблиця, доцільно виділити окремі стовпці для типу тесту, класу еквівалентності або граничного значення. Це полегшить аналіз і зробить звіт більш структурованим.

Під час формування очікуваних результатів необхідно чітко вказувати, що саме повинна зробити система: прийняти дані, відхилити введення, вивести повідомлення про помилку, зберегти запис, заблокувати перехід до наступного кроку тощо. Формулювання на зразок система повинна працювати правильно є недостатнім. Очікуваний результат має бути конкретним і придатним до перевірки.

Після розробки тестових випадків необхідно виконати їх практично або змоделювати виконання, якщо об’єкт тестування є умовним. У процесі виконання слід зафіксувати, які результати отримано фактично, і порівняти їх з очікуваними. Якщо виявлено невідповідності, необхідно коротко описати їх і зазначити, до якого класу або граничного значення вони належать.

На завершальному етапі потрібно проаналізувати достатність побудованих тестів. Слід оцінити, чи охоплено всі основні варіанти введення даних, чи перевірено як допустимі, так і недопустимі значення, чи враховано граничні умови. Також доцільно вказати, які додаткові перевірки можна було б виконати для ще повнішого охоплення функціональності.

Рекомендується не обмежуватися лише найпростішими прикладами. Якщо це можливо, варто розглянути функцію з кількома параметрами, де можна побачити, як поєднання різних еквівалентних класів впливає на кількість тестових випадків. Це допоможе краще зрозуміти практичну цінність методу і його обмеження.

Контрольні питання

1. Що таке тестування типу чорної скриньки?
2. У чому полягає сутність методу еквівалентних класів?
3. Які еквівалентні класи називаються допустимими?
4. Які еквівалентні класи називаються недопустимими?
5. У чому полягає сутність методу граничних значень?
6. Чому помилки часто виникають на межах діапазонів?
7. Які дані повинен містити тестовий випадок?
8. Чому важливо визначати очікуваний результат для кожного тесту?
9. Як пов’язані між собою метод еквівалентних класів і метод граничних значень?
10. Які переваги дає використання методів чорної скриньки під час тестування?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис обраного об’єкта тестування;
– аналіз вимог до вхідних даних;
– виділені еквівалентні класи;
– визначені граничні значення;
– розроблені тестові випадки;
– результати виконання тестів;
– аналіз отриманих результатів;
– висновки.

Практична робота 10. Аналіз коду та покриття

Мета роботи – ознайомитися з методами тестування типу «білої скриньки», виконати аналіз програмного коду із використанням засобів статичного аналізу та оцінити покриття програмного коду.

Ключові положення

Тестування типу «білої скриньки» є підходом до перевірки програмного забезпечення, який базується на аналізі внутрішньої структури програми. У цьому випадку тестувальник

має доступ до вихідного коду і може досліджувати логіку виконання, структуру керування та взаємодію між окремими частинами програми. Такий підхід дозволяє перевірити не лише відповідність результатів, а й правильність реалізації алгоритмів.

Основою тестування «білої скриньки» є аналіз структури програмного коду. Для цього розглядаються послідовності виконання операторів, умовні конструкції, цикли та виклики функцій. Часто використовується уявлення програми у вигляді графа керування, що дозволяє визначити всі можливі шляхи виконання. Це дає змогу систематично формувати тестові випадки для перевірки різних варіантів роботи програми.

Одним із важливих інструментів є статичний аналіз програмного коду. Він передбачає дослідження коду без його виконання і дозволяє виявити помилки, пов'язані з порушенням синтаксису, некоректними конструкціями, потенційними вразливостями та неефективними рішеннями. Статичний аналіз допомагає виявити проблеми на ранніх етапах і зменшити витрати на їх виправлення.

Засоби статичного аналізу можуть автоматично перевіряти код відповідно до заданих правил і стандартів. Вони дозволяють виявляти помилки, які складно знайти під час звичайного тестування, наприклад витоки пам'яті, невикористані змінні або потенційні конфлікти типів. Такі інструменти є важливим доповненням до інших методів забезпечення якості.

Критерії покриття програмного коду визначають ступінь перевірки програми під час тестування. Вони дозволяють оцінити, яка частина коду була виконана і наскільки повно перевірено логіку програми. Найбільш поширеними є покриття операторів, покриття гілок і покриття умов.

Покриття операторів передбачає виконання кожного оператора програми хоча б один раз. Це базовий критерій, який дозволяє перевірити, чи всі частини коду були задіяні під час тестування. Однак він не гарантує перевірку всіх можливих варіантів виконання.

Покриття гілок є більш детальним і передбачає перевірку всіх варіантів умовних переходів. Для кожної умови повинні бути виконані як істинний, так і хибний варіанти. Це дозволяє перевірити правильність логіки розгалужень.

Покриття умов передбачає перевірку всіх логічних виразів, що використовуються у програмі. Особливо важливим це є для складних умов, які містять кілька логічних змінних. Такий критерій дозволяє виявити помилки у логічних конструкціях.

Важливо зазначити, що високий рівень покриття не гарантує відсутності помилок. Він лише свідчить про те, що код був виконаний під час тестування. Тому критерії покриття повинні використовуватися разом із іншими методами аналізу.

Таким чином, тестування типу «білої скриньки» та статичний аналіз програмного коду дозволяють виконати глибоку перевірку внутрішньої структури програмного забезпечення, а критерії покриття дають змогу оцінити повноту тестування.

Практичне завдання

1. Обрати програмний модуль або невеликий програмний проєкт.
2. Проаналізувати структуру програмного коду.
3. Визначити умовні конструкції та цикли.
4. Побудувати схему або опис шляхів виконання.
5. Визначити тестові випадки для покриття операторів.
6. Визначити тестові випадки для покриття гілок.

7. Визначити тестові випадки для покриття умов.
8. Виконати статичний аналіз програмного коду.
9. Зафіксувати виявлені проблеми.
10. Оцінити рівень покриття коду.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з поняттям тестування типу «білої скриньки» та принципами аналізу програмного коду. Особливу увагу слід приділити розумінню логіки виконання програми і побудові можливих шляхів виконання.

На першому етапі необхідно обрати програмний модуль або невеликий проєкт. Рекомендується використовувати код із помірною складністю, що містить умовні конструкції та цикли. Це дозволить продемонструвати застосування критеріїв покриття.

Далі необхідно проаналізувати структуру коду. Слід визначити основні елементи керування, зокрема умови, цикли та виклики функцій. За можливості доцільно представити програму у вигляді логічної схеми або опису шляхів виконання.

Після цього необхідно визначити тестові випадки. Спочатку слід забезпечити покриття операторів, тобто визначити такі вхідні дані, які дозволять виконати всі частини програми. Далі необхідно забезпечити покриття гілок, тобто перевірити всі варіанти умов.

Наступним етапом є визначення тестових випадків для покриття умов. Слід проаналізувати логічні вирази і визначити комбінації значень, які дозволяють перевірити їх.

Паралельно необхідно виконати статичний аналіз коду. Для цього можна використати спеціалізовані інструменти або виконати аналіз вручну. Слід звернути увагу на можливі помилки, неефективні конструкції та порушення стилю.

Після виконання тестування необхідно оцінити рівень покриття. Слід визначити, які частини коду були перевірені і які залишилися непокритими.

На завершальному етапі необхідно проаналізувати результати. Слід визначити, які дефекти були виявлені, і оцінити ефективність використаних методів.

Рекомендується також порівняти результати статичного аналізу і тестування. Це дозволить зрозуміти, які типи помилок виявляються різними методами.

Контрольні питання

1. Що таке тестування типу «білої скриньки»?
2. У чому полягає аналіз структури програмного коду?
3. Що таке статичний аналіз?
4. Які засоби використовуються для статичного аналізу?
5. Що таке покриття операторів?
6. Що таке покриття гілок?
7. Що таке покриття умов?
8. Чому покриття не гарантує відсутності помилок?
9. Які переваги має тестування «білої скриньки»?
10. Як оцінюється рівень покриття коду?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис програмного коду;
– аналіз структури;
– тестові випадки;
– результати статичного аналізу;
– оцінка покриття;
– висновки.

Практична робота 11. Аналіз результатів тестування та дефекти

Мета роботи – навчитися аналізувати результати тестування програмного забезпечення, класифікувати дефекти та оцінювати їх вплив на якість програмного продукту.

Ключові положення

Аналіз результатів тестування є завершальним і водночас одним із найважливіших етапів процесу тестування програмного забезпечення. Саме на цьому етапі здійснюється оцінювання отриманих результатів, визначення відповідності фактичної поведінки системи очікуваній та прийняття рішень щодо подальших дій. Аналіз дозволяє не лише виявити дефекти, а й оцінити загальний рівень якості програмного продукту.

Результати тестування формуються на основі виконання тестових випадків і містять інформацію про проходження або непроходження тестів, виявлені відхилення, а також умови, за яких вони виникли. Важливою умовою є точна фіксація результатів, оскільки це забезпечує можливість їх подальшого аналізу та відтворення ситуацій, у яких виникають дефекти.

Дефект програмного забезпечення є відхиленням фактичної поведінки системи від очікуваної. Дефекти можуть виникати на різних етапах розробки і мати різні причини, зокрема помилки у вимогах, некоректну реалізацію, неправильну інтеграцію або недостатнє тестування. Виявлення і класифікація дефектів дозволяє систематизувати інформацію і підвищити ефективність їх усунення.

Класифікація дефектів може виконуватися за різними ознаками. Однією з основних є класифікація за рівнем критичності. Виділяють критичні дефекти, які роблять систему непридатною до використання, значні дефекти, що впливають на функціональність, та незначні дефекти, які не мають суттєвого впливу на роботу системи.

Іншою важливою ознакою є пріоритет виправлення дефекту. Пріоритет визначає порядок усунення дефектів і залежить від їх впливу на користувача та бізнес-процеси. Наприклад, дефект із високим пріоритетом повинен бути виправлений у першу чергу, навіть якщо його критичність не є максимальною.

Дефекти також можуть класифікуватися за типом, наприклад функціональні помилки, помилки інтерфейсу, помилки продуктивності, логічні помилки або помилки обробки даних. Такий підхід дозволяє виявити проблемні області системи та визначити причини їх виникнення.

Важливим є також аналіз причин виникнення дефектів. Це дозволяє не лише усунути конкретну помилку, а й запобігти появі подібних проблем у майбутньому. Аналіз причин може включати дослідження процесу розробки, перевірку вимог і аналіз коду.

Аналіз результатів тестування також передбачає оцінювання покриття тестами та ефективності тестування. Якщо значна частина дефектів виявляється на пізніх етапах, це може свідчити про недостатню якість тестування на ранніх етапах. Такі висновки дозволяють вдосконалити процес тестування.

Документування результатів тестування є необхідною умовою для подальшого аналізу. Звіти повинні містити інформацію про виконані тести, виявлені дефекти, їх класифікацію та статус. Це забезпечує прозорість процесу і дозволяє відслідковувати зміни у якості програмного забезпечення.

У сучасних проєктах для аналізу результатів тестування використовуються спеціалізовані системи, такі як Jira або Bugzilla, які дозволяють реєструвати дефекти, змінювати їх статус і аналізувати статистику. Це значно підвищує ефективність роботи команди.

Таким чином, аналіз результатів тестування і класифікація дефектів є важливими елементами процесу забезпечення якості, що дозволяють оцінити стан програмного продукту і визначити напрямки його вдосконалення.

Практичне завдання

1. Обрати програмний продукт або модуль для аналізу.
2. Виконати тестування або використати готові результати тестів.
3. Зафіксувати результати виконання тестових випадків.
4. Визначити виявлені дефекти.
5. Класифікувати дефекти за критичністю.
6. Визначити пріоритет виправлення дефектів.
7. Класифікувати дефекти за типами.
8. Проаналізувати причини виникнення дефектів.
9. Оцінити ефективність тестування.
10. Сформувати загальний висновок.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з поняттям дефекту програмного забезпечення та принципами аналізу результатів тестування. Особливу увагу слід приділити класифікації дефектів і визначенню їх критичності.

На першому етапі необхідно обрати програмний продукт або використати результати попередніх практичних робіт. Це дозволить забезпечити логічну послідовність виконання завдань і використати вже отримані дані.

Далі необхідно виконати тестування або проаналізувати вже наявні результати. Слід зафіксувати, які тестові випадки були виконані і які результати отримані. Важливо забезпечити точність і повноту записів.

Після цього необхідно визначити дефекти. Для кожного дефекту слід описати умови виникнення, фактичний результат і очікувану поведінку системи. Це дозволить правильно класифікувати дефекти.

Наступним етапом є класифікація дефектів. Слід визначити їх критичність, пріоритет і тип. Це дозволить оцінити вплив дефектів на якість програмного забезпечення.

Далі необхідно проаналізувати причини виникнення дефектів. Слід визначити, на якому етапі розробки виникла помилка і які фактори на це вплинули.

Після цього необхідно оцінити ефективність тестування. Слід визначити, чи були виявлені всі суттєві дефекти і чи достатнім було покриття тестами.

За можливості доцільно використати систему відстеження дефектів, наприклад Jira. Це дозволить краще організувати процес аналізу.

На завершальному етапі необхідно сформувати висновки. Слід оцінити якість програмного забезпечення і визначити напрямки його вдосконалення.

Контрольні питання

1. Що таке результат тестування?
2. Що таке дефект програмного забезпечення?
3. Які види дефектів існують?
4. Що таке критичність дефекту?
5. Що таке пріоритет дефекту?
6. Як класифікуються дефекти?
7. Чому важливо аналізувати причини дефектів?
8. Як оцінюється ефективність тестування?
9. Яку роль відіграє документування результатів?
10. Які інструменти використовуються для відстеження дефектів?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис програмного продукту;
– результати тестування;
– класифікація дефектів;
– аналіз причин;
– оцінка ефективності тестування;
– висновки.

ТЕМА 5. АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ ТА ОЦІНЮВАННЯ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Практична робота 12. Автоматизація тестування

Мета роботи – ознайомитися із застосуванням інструментальних засобів автоматизації тестування, навчитися створювати та виконувати автоматизовані тести для програмного забезпечення.

Ключові положення

Автоматизація тестування програмного забезпечення є підходом, що передбачає використання спеціалізованих програмних засобів для виконання тестових перевірок без постійної участі людини. Вона дозволяє значно підвищити ефективність процесу тестування, забезпечити повторюваність перевірок і скоротити час виконання великої кількості тестових сценаріїв. У сучасних проєктах автоматизація є невід'ємною складовою процесу розробки.

Інструментальні засоби автоматизації тестування дозволяють створювати тестові сценарії у вигляді програмного коду або конфігурацій, виконувати їх у заданому середовищі, аналізувати результати та формувати звіти. Такі засоби можуть застосовуватися на різних рівнях тестування, включаючи модульне, інтеграційне та системне тестування.

Для модульного тестування широко використовуються фреймворки, такі як JUnit та pytest. Вони дозволяють перевіряти окремі функції або класи, автоматично запускати тести та оцінювати результати. Це забезпечує швидке виявлення помилок на ранніх етапах розробки.

Для тестування вебзастосунків застосовуються інструменти, такі як Selenium, який дозволяє автоматизувати взаємодію з вебінтерфейсом, і Cypress, що забезпечує виконання тестів у браузері з високою швидкістю. Такі засоби дозволяють перевіряти поведінку інтерфейсу користувача та взаємодію з серверною частиною.

Автоматизація тестування особливо ефективна для виконання регресійних перевірок. При внесенні змін у програмний код необхідно переконатися, що існуюча функціональність не порушена. Автоматизовані тести можуть виконуватися при кожній зміні, що дозволяє оперативно виявляти дефекти.

Важливою складовою є інтеграція автоматизованого тестування у процеси безперервної інтеграції. Для цього використовуються системи, такі як Jenkins, які дозволяють автоматично запускати тести після внесення змін у код. Це забезпечує постійний контроль якості програмного забезпечення.

Разом із тим автоматизація тестування має обмеження. Створення автоматизованих тестів потребує значних початкових витрат, а також постійної підтримки. Крім того, не всі типи тестування доцільно автоматизувати, зокрема ті, що пов'язані з оцінюванням зручності використання або візуального сприйняття.

Ефективність автоматизації залежить від правильного вибору тестів, які доцільно автоматизувати. Найбільшу користь вона приносить у випадках повторюваних перевірок, стабільної функціональності та необхідності швидкого отримання результатів.

Таким чином, інструментальні засоби автоматизації тестування дозволяють підвищити ефективність перевірки програмного забезпечення, забезпечити стабільність процесу тестування та зменшити вплив людського фактора.

Практичне завдання

1. Обрати програмний продукт або модуль для тестування.
2. Визначити функціональність, що буде автоматизована.
3. Обрати інструмент автоматизації тестування.
4. Встановити обраний інструмент.
5. Розробити автоматизовані тестові сценарії.
6. Виконати автоматизовані тести.
7. Проаналізувати результати виконання тестів.
8. Зафіксувати виявлені дефекти.
9. Оцінити ефективність автоматизації.
10. Сформувати висновки.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з принципами автоматизації тестування та особливостями використання відповідних інструментів. Особливу увагу слід приділити вибору інструменту залежно від типу програмного забезпечення.

На першому етапі необхідно обрати програмний продукт або модуль. Рекомендується використовувати невеликий застосунок, що дозволить швидко створити і виконати автоматизовані тести.

Далі необхідно визначити функціональність, яка буде автоматизована. Доцільно обрати повторювані операції, які часто виконуються під час тестування.

Після цього необхідно обрати інструмент автоматизації, наприклад pytest або Selenium. Вибір залежить від типу застосунку.

Наступним етапом є встановлення інструменту і налаштування середовища. Слід перевірити працездатність інструменту перед початком роботи.

Далі необхідно розробити тестові сценарії. Вони повинні містити перевірки основної функціональності системи. Особливу увагу слід приділити правильності формування очікуваних результатів.

Після цього необхідно виконати автоматизовані тести. Результати виконання слід зафіксувати і проаналізувати.

У разі виявлення дефектів необхідно описати їх і визначити причини виникнення. Це дозволить оцінити ефективність тестування.

На завершальному етапі необхідно оцінити доцільність використання автоматизації. Слід визначити, які переваги вона дає і які обмеження має.

Рекомендується також порівняти автоматизоване тестування з ручним і визначити їх особливості.

Контрольні питання

1. Що таке автоматизоване тестування?
2. Які переваги автоматизації тестування?
3. Які обмеження має автоматизація?
4. Які інструменти використовуються для автоматизації?
5. Для чого використовується JUnit?
6. Для чого використовується Selenium?
7. Що таке регресійне тестування?
8. Як інтегрується автоматизація у процес розробки?
9. Які тести доцільно автоматизувати?
10. Як оцінюється ефективність автоматизації?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис програмного продукту;
– опис автоматизованих тестів;
– результати виконання;
– аналіз ефективності;
– висновки.

Практична робота 13. Управління тестуванням та дефектами

Мета роботи – ознайомитися із застосуванням систем управління тестуванням та систем відстеження дефектів, навчитися організовувати тестову діяльність і фіксувати дефекти програмного забезпечення.

Ключові положення

Системи управління тестуванням є інструментами, що забезпечують організацію, планування і контроль процесу тестування програмного забезпечення. Вони дозволяють централізовано зберігати тестові сценарії, тестові випадки, результати виконання тестів і пов’язану документацію. Використання таких систем є особливо важливим у командних проєктах, де необхідно забезпечити узгодженість дій та прозорість процесу.

Основною функцією систем управління тестуванням є структуризація тестових матеріалів. Тестові випадки групуються за функціональністю, пріоритетами або етапами тестування. Це дозволяє ефективно планувати виконання тестів і контролювати їх покриття. Крім того, такі системи забезпечують можливість відслідковування стану виконання тестів і формування звітності.

До поширених систем управління тестуванням належать TestRail, Zephyr та qTest. Вони надають інструменти для створення тестових наборів, планування виконання тестів і аналізу результатів.

Системи відстеження дефектів призначені для реєстрації, класифікації та контролю процесу виправлення помилок у програмному забезпеченні. Вони дозволяють фіксувати дефекти, визначати їх критичність і пріоритет, а також відслідковувати статус виправлення. Це забезпечує системний підхід до управління якістю.

Життєвий цикл дефекту включає кілька станів, таких як створення, підтвердження, виправлення, перевірка та закриття. Кожен дефект проходить через ці етапи, що дозволяє контролювати процес його обробки. Наявність чітко визначених станів забезпечує прозорість і контроль.

Серед найбільш поширених систем відстеження дефектів використовуються Jira, Bugzilla та Redmine. Вони дозволяють ефективно організувати роботу з дефектами і забезпечити взаємодію між учасниками проєкту.

Важливим є зв'язок між тестовими випадками і дефектами. Кожен виявлений дефект повинен бути пов'язаний із відповідним тестом, що дозволяє відслідковувати, які перевірки призвели до його виявлення. Це забезпечує трасованість і підвищує ефективність аналізу.

Інтеграція систем управління тестуванням і відстеження дефектів дозволяє створити єдине середовище для контролю якості. Це забезпечує узгодженість процесів, скорочує час на обробку інформації та підвищує ефективність роботи команди.

Таким чином, використання систем управління тестуванням і відстеження дефектів є важливим елементом сучасного процесу розробки програмного забезпечення, що дозволяє підвищити якість і керованість проєкту.

Практичне завдання

1. Обрати програмний проєкт для тестування.
2. Обрати систему управління тестуванням.
3. Створити тестовий проєкт у системі.
4. Додати тестові сценарії і тестові випадки.
5. Сформувати тестовий набір.
6. Виконати тестування і зафіксувати результати.
7. Виявити дефекти.
8. Обрати систему відстеження дефектів.
9. Зареєструвати дефекти у системі.
10. Проаналізувати процес обробки дефектів.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з принципами роботи систем управління тестуванням і систем відстеження дефектів. Особливу увагу слід приділити їх ролі у процесі забезпечення якості програмного забезпечення.

На першому етапі необхідно обрати програмний проєкт. Рекомендується використовувати проєкт, для якого вже розроблені тестові випадки, наприклад із попередніх практичних робіт. Це дозволить зосередитися на організації процесу тестування.

Далі необхідно обрати систему управління тестуванням, наприклад TestRail або Zephyr. У вибраній системі слід створити тестовий проєкт і додати тестові випадки.

Після цього необхідно сформувати тестові набори і виконати тестування. Результати виконання слід зафіксувати у системі. Важливо правильно відмітити статус кожного тесту.

На наступному етапі необхідно виявити дефекти і зареєструвати їх у системі відстеження дефектів, наприклад Jira. Для кожного дефекту слід вказати опис, умови виникнення і пріоритет.

Далі необхідно проаналізувати процес обробки дефектів. Слід відслідкувати зміну статусів і визначити, як відбувається їх виправлення.

На завершальному етапі необхідно оцінити ефективність використання систем. Слід визначити їх переваги і можливі обмеження.

Рекомендується також проаналізувати взаємодію між системами управління тестуванням і відстеження дефектів. Це дозволить краще зрозуміти організацію процесу тестування.

Контрольні питання

1. Що таке система управління тестуванням?
2. Які функції вона виконує?
3. Що таке система відстеження дефектів?
4. Які етапи має життєвий цикл дефекту?
5. Яку роль відіграє TestRail?
6. Яку роль відіграє Jira?
7. Що таке трасованість тестів?
8. Як реєструються дефекти?
9. Як аналізуються результати тестування?
10. Чому важливо використовувати спеціалізовані системи?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис програмного проєкту;
– опис системи управління тестуванням;
– результати виконання тестів;
– опис дефектів;
– аналіз процесу обробки;
– висновки.

Практична робота 14. Оцінювання якості за метриками

Мета роботи – навчитися оцінювати якість програмного забезпечення з використанням метрик, виконувати кількісний аналіз та формувати обґрунтовані висновки.

Ключові положення

Оцінювання якості програмного забезпечення з використанням метрик є важливим етапом аналізу програмних систем, що дозволяє перейти від описових характеристик до кількісних показників. Метрики забезпечують можливість об’єктивного оцінювання

властивостей програмного продукту, порівняння різних рішень і визначення тенденцій зміни якості у процесі розробки.

Метрики програмного забезпечення відображають різні характеристики, зокрема складність, надійність, продуктивність, супроводжуваність і рівень тестування. Вони можуть застосовуватися як до програмного продукту, так і до процесу його розробки. Використання метрик дозволяє виявляти проблемні області та визначати напрямки вдосконалення.

Однією з ключових груп є метрики складності. Вони дозволяють оцінити структуру програмного коду і визначити рівень його складності. Зростання складності зазвичай призводить до підвищення ймовірності помилок і ускладнення супроводу. Тому контроль складності є важливим для забезпечення якості.

Метрики дефектів є важливим показником надійності програмного забезпечення. До них належать кількість дефектів, щільність дефектів і час їх усунення. Аналіз цих показників дозволяє оцінити стабільність системи і ефективність процесу тестування.

Метрики продуктивності характеризують ефективність роботи програмного забезпечення. До них належать час відгуку, пропускна здатність і використання ресурсів. Вони дозволяють оцінити, наскільки ефективно система виконує свої функції у різних умовах.

Метрики покриття тестами визначають ступінь перевірки програмного коду. Високий рівень покриття свідчить про більш повну перевірку, але не гарантує відсутності дефектів. Тому ці метрики повинні використовуватися разом з іншими показниками.

Важливою особливістю є необхідність комплексного використання метрик. Окрема метрика не може повністю характеризувати якість програмного забезпечення. Тільки сукупність показників дозволяє отримати об’єктивну оцінку.

Під час аналізу метрик необхідно враховувати контекст проєкту, тип програмного забезпечення і вимоги до якості. Це дозволяє правильно інтерпретувати отримані значення і уникнути помилкових висновків.

Таким чином, використання метрик програмного забезпечення дозволяє виконувати кількісне оцінювання якості, виявляти проблемні області і приймати обґрунтовані рішення щодо вдосконалення програмного продукту.

Практичне завдання

1. Обрати програмний продукт або програмний модуль.
2. Визначити перелік метрик для оцінювання.
3. Оцінити розмір програмного забезпечення.
4. Визначити складність програмного коду.
5. Зібрати дані про дефекти.
6. Обчислити щільність дефектів.
7. Оцінити продуктивність системи.
8. Визначити рівень покриття тестами.
9. Проаналізувати отримані показники.
10. Сформувати висновок щодо якості програмного забезпечення.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з основними метриками програмного забезпечення та їх призначенням. Особливу увагу слід приділити розумінню того, які метрики відповідають за оцінювання різних характеристик якості.

На першому етапі необхідно обрати програмний продукт або модуль. Рекомендується використовувати програму, для якої доступні дані про код і результати тестування. Це дозволить виконати більш точний аналіз.

Далі необхідно визначити перелік метрик. Слід обрати показники, що дозволяють оцінити різні властивості системи. Важливо забезпечити баланс між різними характеристиками.

Після цього необхідно виконати обчислення метрик. Для цього можна використати як ручні розрахунки, так і спеціалізовані інструменти. Слід забезпечити точність отриманих даних.

Під час аналізу результатів необхідно оцінити значення метрик і визначити, які з них свідчать про проблеми. Слід звернути увагу на взаємозв’язок показників.

Важливо також врахувати обмеження метрик. Слід розуміти, що окремі показники не дають повної картини якості.

На завершальному етапі необхідно сформувати висновки. Слід оцінити якість програмного забезпечення і визначити напрямки його вдосконалення.

Рекомендується також порівняти отримані результати з очікуваними значеннями або стандартами.

Контрольні питання

1. Що таке метрики програмного забезпечення?
2. Які види метрик існують?
3. Що характеризують метрики складності?
4. Як визначається щільність дефектів?
5. Які метрики використовуються для оцінювання продуктивності?
6. Що таке покриття тестами?
7. Чому важливо використовувати кілька метрик?
8. Які обмеження мають метрики?
9. Як аналізуються результати метрик?
10. Як метрики використовуються для оцінювання якості?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис програмного продукту;
– перелік метрик;
– результати обчислень;
– аналіз показників;
– висновки.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Основна

1. Ammann, Paul, Offutt, Jeff. Introduction to Software Testing. 2nd ed. Cambridge University Press, 2017. 345 p. ISBN 9781107172012.
2. Spillner, A., Linz, T. Software Testing Foundations: A Study Guide for the Certified Tester Exam- Foundation Level- ISTQB Compliant. Dpunkt.Verlag. 2021. 339 p. ISBN: 9783969102992, 3969102995
3. Lewis, William E. Software Testing and Continuous Quality Improvement. 3rd ed. CRC Press, 2017. 688 p. ISBN: 9781439834367, 1439834369.
4. Авраменко А. С. Тестування програмного забезпечення : навчальний посібник. Черкаси : Черкаський національний університет ім. Б. Хмельницького, 2017. 208 с. ISBN 9789669201997.
5. Трофименко О. Г. Тестування та забезпечення якості програмних систем : навчально-методичний посібник [Електронне видання]. Одеса : Фенікс, 2024. 140 с.

Допоміжна

6. Myers, Glenford J., Sandler, Corey, Badgett, Tom. The Art of Software Testing. 3rd ed. John Wiley & Sons, 2011. 256 p. ISBN: 9781118133156, 1118133153.
7. Jorgensen, P. C., DeVries, B. Software Testing: A Craftsman's Approach, Fifth Edition. CRC Press. 2021. 550 p. ISBN: 9781000391527, 1000391523.
8. Gregory, J., Crispin, L. The Agile Testing Collection. Pearson Education. 2015. 1114 p. ISBN: 9780134190631, 0134190637.
9. Software Engineering for Agile Application Development. IGI Global. 2020. 330 p. ISBN: 9781799825333, 1799825337
10. Скорін Ю. І. Тестування програмного забезпечення : навчально-методичний посібник. Харків : ХНЕУ ім. С. Кузнеця, 2022. 47 с.

Інформаційні ресурси

11. International Software Testing Qualifications Board (ISTQB). URL: https://www.istqb.org/
12. Selenium – Web Browser Automation. URL: https://www.selenium.dev/
13. JUnit – Unit Testing Framework for Java. URL: https://junit.org/
14. SonarQube – Platform for Continuous Code Quality Inspection. URL: https://www.sonarsource.com/products/sonarqube/
15. Automation Testing Tutorial. Guru99. URL: https://www.guru99.com/software-testing.html

Навчальне видання

Сергій Борисович Приходько

ЯКІСТЬ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Методичні вказівки до практичних робіт для здобувачів вищої освіти,
які навчаються за спеціальностями галузі «Інформаційні технології»

Українською мовою