Міністерство освіти і науки України
Міжнародний гуманітарний університет
Кафедра комп'ютерної інженерії та інноваційних технологій

О.Ю. Лебедєва

ТЕХНОЛОГІЇ СТВОРЕННЯ ПРОГРАМНИХ ПРОДУКТІВ

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.

Лебедєва О.Ю. Технології створення програмних продуктів. 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. 64 с.

Дисципліна «Технології створення програмних продуктів» спрямована на формування системного розуміння процесів розробки програмного забезпечення від етапу формування вимог до введення програмного продукту в експлуатацію та його супроводження. Курс охоплює моделі життєвого циклу програмного забезпечення, сучасні методології розробки, принципи архітектурного проєктування, компонентний підхід, використання систем контролю версій та інструментальних засобів автоматизації. Значна увага приділяється забезпеченню якості програмного забезпечення, тестуванню, верифікації та валідації, а також управлінню змінами та інтеграції програмних компонентів.

ЗМІСТ

ТЕМА 1. ЕТАПИ ЖИТТЄВОГО ЦИКЛУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Практична робота 1. Аналіз прикладу програмного продукту та визначення моделі життєвого циклу

Мета роботи – ознайомитися з основними моделями життєвого циклу програмного забезпечення, проаналізувати етапи розробки програмного продукту та визначити модель життєвого циклу для конкретного прикладу.

Ключові положення

Життєвий цикл програмного забезпечення є сукупністю процесів, що охоплюють усі етапи існування програмного продукту від моменту формування ідеї до завершення його використання. Він визначає порядок виконання робіт, взаємодію між учасниками процесу розробки та підходи до управління проєктом.

Основними етапами життєвого циклу програмного забезпечення є формування вимог, проєктування, реалізація, тестування, впровадження та супроводження. На етапі формування вимог визначаються функціональні та нефункціональні характеристики програмного продукту. Проєктування передбачає створення архітектури системи та визначення структури її компонентів. Реалізація полягає у програмній розробці та інтеграції окремих модулів. Тестування забезпечує перевірку відповідності програмного продукту встановленим вимогам. Впровадження передбачає введення системи в експлуатацію, а супроводження — підтримку її працездатності та подальший розвиток.

Існує декілька моделей життєвого циклу програмного забезпечення, які визначають порядок проходження зазначених етапів. Найбільш поширеними є каскадна, ітераційна та спіральна моделі.

Каскадна модель передбачає послідовне виконання етапів, де кожен наступний етап починається після завершення попереднього. Такий підхід є зрозумілим і структурованим, однак має обмежену гнучкість у разі змін вимог.

Ітераційна модель базується на багаторазовому повторенні етапів розробки з поступовим уточненням функціональності програмного продукту. Вона дозволяє враховувати зміни вимог і поступово вдосконалювати систему.

Спіральна модель поєднує ітераційний підхід із аналізом ризиків. Кожен виток спіралі відповідає певному набору етапів розробки, що супроводжуються оцінюванням можливих ризиків та їх мінімізацією.

Вибір моделі життєвого циклу залежить від складності проєкту, визначеності вимог, обмежень за часом та ресурсами. Для невеликих проєктів із чітко визначеними вимогами доцільним є використання каскадної моделі. У випадку складних систем із змінними вимогами більш ефективними є ітераційні підходи.

Таким чином, аналіз життєвого циклу програмного забезпечення дозволяє обґрунтовано обрати підхід до розробки програмного продукту та забезпечити ефективну організацію процесу його створення.

Практичне завдання

1. Ознайомитися з поняттям життєвого циклу програмного забезпечення.
2. Проаналізувати основні етапи життєвого циклу програмного продукту.
3. Ознайомитися з каскадною моделлю життєвого циклу.
4. Ознайомитися з ітераційною моделлю життєвого циклу.
5. Ознайомитися зі спіральною моделлю життєвого циклу.
6. Обрати приклад програмного продукту для аналізу.
7. Визначити основні етапи розробки обраного програмного продукту.
8. Проаналізувати особливості організації процесу розробки.
9. Визначити модель життєвого циклу, що найбільше відповідає обраному прикладу.
10. Обґрунтувати вибір моделі життєвого циклу.
11. Зробити висновки щодо ефективності обраної моделі.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з основними моделями життєвого циклу програмного забезпечення. Особливу увагу слід приділити відмінностям між послідовними та ітераційними підходами до розробки.

На першому етапі необхідно обрати програмний продукт для аналізу. Це може бути вебзастосунок, мобільний застосунок або інша програмна система. Доцільно обрати приклад, структура якого є достатньо зрозумілою для аналізу.

Далі необхідно визначити етапи розробки обраного програмного продукту. Слід проаналізувати, як формувалися вимоги, яким чином здійснювалося проєктування та реалізація, а також як проводилося тестування та впровадження.

Особливу увагу необхідно приділити послідовності виконання етапів. Слід визначити, чи виконувалися етапи один раз послідовно, чи відбувалося їх повторення у вигляді ітерацій.

На наступному етапі потрібно співвіднести отримані результати з відомими моделями життєвого циклу. Необхідно визначити, яка модель найбільш точно відображає процес розробки обраного програмного продукту.

У процесі аналізу доцільно враховувати наявність змін вимог, частоту випуску нових версій, підходи до тестування та управління ризиками.

Після визначення моделі життєвого циклу необхідно обґрунтувати свій вибір. Пояснення повинно базуватися на аналізі реальних характеристик процесу розробки.

Після виконання роботи необхідно зробити висновки щодо доцільності використання обраної моделі життєвого циклу у подібних проєктах.

Контрольні питання

1. Що таке життєвий цикл програмного забезпечення?
2. Які основні етапи життєвого циклу програмного продукту?
3. У чому полягає сутність каскадної моделі?
4. Які особливості ітераційної моделі?
5. У чому полягає сутність спіральної моделі?
6. Які фактори впливають на вибір моделі життєвого циклу?
7. Які переваги та недоліки каскадної моделі?

8. Які переваги ітераційного підходу?
9. Яку роль відіграє аналіз ризиків у спіральній моделі?
10. Як визначити модель життєвого циклу для програмного продукту?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– короткий опис моделей життєвого циклу програмного забезпечення;
– характеристика обраного програмного продукту;
– опис етапів його розробки;
– визначення моделі життєвого циклу;
– обґрунтування вибору моделі;
– висновки.

Практична робота 2. Формування функціональних і нефункціональних вимог до програмної системи

Мета роботи – ознайомитися з підходами до формування вимог до програмного забезпечення, навчитися визначати функціональні та нефункціональні вимоги та формалізувати їх для програмної системи.

Ключові положення

Вимоги до програмного забезпечення визначають очікувану поведінку системи, її функціональні можливості та обмеження, які повинні бути враховані під час розробки. Вони є основою для проєктування, реалізації та тестування програмного продукту.

Функціональні вимоги описують, які саме функції повинна виконувати система. Вони визначають поведінку системи у відповідь на вхідні дані та взаємодію з користувачем або іншими системами. До функціональних вимог належать операції обробки даних, введення та виведення інформації, виконання обчислень, формування звітів та інші дії, що реалізуються програмною системою.

Нефункціональні вимоги визначають характеристики якості програмного забезпечення та обмеження його функціонування. Вони описують умови, за яких система повинна працювати, і включають вимоги до продуктивності, надійності, безпеки, зручності використання, масштабованості та сумісності. Такі вимоги не визначають конкретні функції, але суттєво впливають на архітектуру системи.

Формування вимог передбачає їх виявлення, аналіз, документування та узгодження із зацікавленими сторонами. Важливо забезпечити повноту, однозначність і несуперечливість вимог. Вимоги повинні бути сформульовані таким чином, щоб їх можна було перевірити під час тестування.

Для опису вимог часто використовуються текстові специфікації, сценарії використання, діаграми та формальні моделі. Доцільно застосовувати стандартизовані підходи до документування вимог, що дозволяє забезпечити їх зрозумілість і зручність використання у процесі розробки.

Якість сформованих вимог безпосередньо впливає на успішність реалізації програмного продукту. Некоректно визначені вимоги можуть призвести до помилок у програмі, перевитрат ресурсів та необхідності повторної розробки окремих компонентів.

Таким чином, формування функціональних і нефункціональних вимог є ключовим етапом розробки програмного забезпечення, що визначає подальший процес створення програмної системи.

Практичне завдання

1. Ознайомитися з поняттям вимог до програмного забезпечення.
2. Визначити відмінності між функціональними та нефункціональними вимогами.
3. Обрати приклад програмної системи для аналізу.
4. Сформувати перелік функціональних вимог для обраної системи.
5. Сформувати перелік нефункціональних вимог.
6. Визначити вимоги до продуктивності програмної системи.
7. Визначити вимоги до надійності та безпеки.
8. Визначити вимоги до зручності використання та сумісності.
9. Перевірити сформовані вимоги на повноту та несуперечливість.
10. Оформити результати у вигляді специфікації вимог.
11. Зробити висновки щодо якості сформованих вимог.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з основами формування вимог до програмного забезпечення. Доцільно звернути увагу на класифікацію вимог та їх роль у процесі розробки програмних систем.

На першому етапі потрібно обрати програмну систему для аналізу. Це може бути вебзастосунок, мобільний застосунок або інша інформаційна система. Важливо, щоб система мала достатню функціональність для формування повного набору вимог.

Далі необхідно сформувати функціональні вимоги. Слід описати основні дії, які повинна виконувати система, а також взаємодію користувача із програмним продуктом. Доцільно використовувати чіткі та однозначні формулювання.

Після цього потрібно визначити нефункціональні вимоги. Необхідно врахувати характеристики продуктивності, надійності, безпеки, зручності використання та сумісності. Вимоги повинні бути конкретними та вимірюваними.

У процесі формування вимог необхідно перевірити їх на повноту та узгодженість. Слід уникати суперечностей між різними вимогами та забезпечити їх логічну відповідність.

Особливу увагу необхідно приділити можливості перевірки вимог. Кожна вимога повинна бути сформульована таким чином, щоб її виконання можна було підтвердити під час тестування.

Після формування вимог необхідно оформити їх у вигляді структурованого документа. Доцільно використовувати уніфіковану форму подання вимог.

Після виконання роботи необхідно зробити висновки щодо процесу формування вимог та їх впливу на якість програмного забезпечення.

Контрольні питання

1. Що таке вимоги до програмного забезпечення?
2. Які існують види вимог?
3. У чому полягає відмінність між функціональними та нефункціональними вимогами?
4. Які приклади функціональних вимог можна навести?
5. Які характеристики описують нефункціональні вимоги?
6. Що таке вимоги до продуктивності?
7. Які вимоги відносяться до безпеки програмного забезпечення?
8. Чому важлива перевірюваність вимог?
9. Які помилки можуть виникати під час формування вимог?
10. Як оформлюється специфікація вимог?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис обраної програмної системи;
– перелік функціональних вимог;
– перелік нефункціональних вимог;
– аналіз якості сформованих вимог;
– оформлена специфікація вимог;
– висновки.

Практична робота 3. Побудова специфікації вимог та таблиці трасування

Мета роботи – ознайомитися з принципами побудови специфікації вимог до програмного забезпечення, навчитися структурувати вимоги та формувати таблицю трасування для забезпечення їх повноти і контрольованості.

Ключові положення

Специфікація вимог до програмного забезпечення є базовим документом у процесі розробки програмного продукту, який визначає всі суттєві характеристики системи та слугує основою для подальших етапів життєвого циклу. Вона використовується як джерело інформації для розробників, тестувальників, аналітиків та інших учасників проєкту, забезпечуючи узгоджене розуміння функціонування програмної системи.

Специфікація вимог повинна містити як функціональні, так і нефункціональні вимоги. Функціональні вимоги описують поведінку системи, її реакцію на вхідні дані та взаємодію із зовнішніми об’єктами. Нефункціональні вимоги визначають характеристики якості, такі як продуктивність, надійність, безпека, масштабованість і зручність використання.

Якість специфікації вимог безпосередньо впливає на ефективність розробки програмного забезпечення. Нечіткі або суперечливі вимоги можуть призвести до помилок на етапі реалізації, що, у свою чергу, спричиняє додаткові витрати часу та ресурсів. Тому

важливо забезпечити однозначність формулювань, уникати двозначностей та використовувати чіткі критерії перевірки.

Однією з ключових характеристик якісних вимог є їх перевірюваність. Це означає, що для кожної вимоги повинні бути визначені критерії, які дозволяють встановити факт її виконання. Такі критерії зазвичай формулюються у вигляді тестових умов або сценаріїв перевірки.

Структура специфікації вимог зазвичай включає вступ, загальний опис системи, визначення термінів, перелік функціональних вимог, нефункціональні вимоги, обмеження, припущення та додаткові матеріали. У складних проєктах також можуть використовуватися діаграми, що відображають взаємодію між компонентами системи.

Важливим елементом специфікації є ідентифікація вимог. Кожна вимога повинна мати унікальний ідентифікатор, що дозволяє однозначно посилатися на неї у процесі розробки, тестування та супроводження програмного забезпечення. Використання ідентифікаторів забезпечує зручність керування вимогами та їх змінами.

Трасування вимог є процесом встановлення зв’язків між вимогами та іншими артефактами розробки. До таких артефактів належать проєктні рішення, програмні модулі, тестові випадки та документація. Трасування дозволяє визначити, яким чином кожна вимога реалізується у системі та як вона перевіряється.

Таблиця трасування вимог є інструментом, що дозволяє систематизувати інформацію про зв’язки між вимогами та іншими елементами системи. Вона зазвичай містить ідентифікатор вимоги, її короткий опис, відповідний модуль або компонент системи, а також тестові сценарії, що перевіряють виконання вимоги.

Використання таблиці трасування забезпечує можливість контролю повноти реалізації вимог. Це дозволяє виявити вимоги, які не були реалізовані або не мають відповідних тестів, а також уникнути надлишкових функцій, що не пов’язані з вимогами.

Крім того, трасування вимог відіграє важливу роль у процесі управління змінами. У разі зміни вимоги можна швидко визначити, які компоненти системи необхідно модифікувати, що дозволяє мінімізувати ризики та скоротити витрати на внесення змін.

Таким чином, побудова специфікації вимог і таблиці трасування є важливими складовими процесу розробки програмного забезпечення, що забезпечують структурованість, контрольованість і якість програмного продукту.

Практичне завдання

1. Ознайомитися з поняттям специфікації вимог до програмного забезпечення.
2. Ознайомитися з основними характеристиками якісних вимог.
3. Обрати приклад програмної системи.
4. Сформувати перелік функціональних вимог.
5. Сформувати перелік нефункціональних вимог.
6. Присвоїти кожній вимозі унікальний ідентифікатор.
7. Оформити специфікацію вимог у структурованому вигляді.
8. Побудувати таблицю трасування вимог.
9. Встановити зв’язки між вимогами та елементами системи.
10. Встановити зв’язки між вимогами та тестовими сценаріями.
11. Проаналізувати повноту покриття вимог.
12. Зробити висновки щодо якості сформованої специфікації.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно детально ознайомитися з принципами документування вимог до програмного забезпечення. Доцільно звернути увагу на стандартизацію підходів до опису вимог та використання уніфікованих шаблонів специфікацій.

На першому етапі необхідно обрати програмну систему, для якої буде створено специфікацію вимог. Вибір системи повинен бути обґрунтованим, а її функціональність – достатньо широкою для формування повного набору вимог. Доцільно обирати системи, що мають як користувацькі, так і адміністративні функції.

Далі необхідно сформувати перелік функціональних вимог. Для цього слід визначити основні сценарії використання системи та описати дії користувачів. Вимоги повинні бути сформульовані чітко, без двозначностей, із зазначенням очікуваного результату виконання кожної функції.

Після цього потрібно визначити нефункціональні вимоги. Доцільно розглянути різні категорії таких вимог, зокрема продуктивність, безпеку, надійність, зручність використання, сумісність та масштабованість. Кожна вимога повинна бути конкретною та, за можливості, містити кількісні показники.

Особливу увагу необхідно приділити ідентифікації вимог. Ідентифікатори повинні бути унікальними та мати логічну структуру, що дозволяє легко орієнтуватися у переліку вимог. Наприклад, можна використовувати префікси для позначення типу вимоги.

На наступному етапі необхідно оформити специфікацію вимог у вигляді структурованого документа. Вимоги доцільно групувати за категоріями, забезпечуючи логічну послідовність викладу матеріалу.

Далі необхідно побудувати таблицю трасування вимог. Для цього слід створити таблицю, у якій кожній вимозі відповідають елементи реалізації та тестові сценарії. Така таблиця дозволяє встановити зв’язки між різними артефактами розробки.

У процесі побудови таблиці трасування необхідно перевірити, щоб кожна вимога була пов’язана з відповідним модулем або компонентом системи, а також мала хоча б один тестовий сценарій для перевірки її виконання.

Особливу увагу слід приділити аналізу повноти покриття вимог. Усі вимоги повинні бути реалізовані та перевірені. У разі виявлення вимог, які не мають відповідних зв’язків, необхідно доопрацювати специфікацію або таблицю трасування.

Після завершення роботи доцільно провести аналіз отриманих результатів. Необхідно оцінити якість сформованої специфікації, зручність використання таблиці трасування та можливість її застосування для управління змінами.

Таким чином, виконання роботи дозволяє сформувати практичні навички документування вимог і забезпечення їх повного покриття у процесі розробки програмного забезпечення.

Контрольні питання

1. Що таке специфікація вимог до програмного забезпечення?
2. Які характеристики повинні мати якісні вимоги?
3. Яка структура специфікації вимог?
4. Для чого використовуються ідентифікатори вимог?

5. Що таке трасування вимог?
6. Яка мета побудови таблиці трасування?
7. Які елементи можуть входити до таблиці трасування?
8. Як забезпечується повнота покриття вимог?
9. Яким чином таблиця трасування допомагає керувати змінами?
10. Які переваги використання специфікації вимог?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис обраної програмної системи;
– специфікація вимог;
– таблиця трасування вимог;
– аналіз покриття вимог;
– висновки.

Практична робота 4. Планування ітерації розробки за підходом Scrum

Мета роботи – ознайомитися з принципами гнучкої розробки програмного забезпечення за підходом Scrum, навчитися формувати беклог продукту, визначати обсяг робіт і планувати ітерацію розробки.

Ключові положення

Scrum є гнучким підходом до організації процесу розробки програмного забезпечення, що базується на ітераційному та інкрементальному створенні продукту. Основною ідеєю є поступове нарощування функціональності системи через виконання коротких ітерацій, які дозволяють регулярно отримувати проміжні результати та коригувати подальший напрям розробки.

Ітерація у Scrum називається спринтом і має фіксовану тривалість. У межах одного спринту команда виконує визначений обсяг робіт, що має завершитися створенням працездатного інкременту програмного продукту. Інкремент є частиною системи, яка відповідає визначеним вимогам і може бути продемонстрована замовнику.

Scrum передбачає чіткий розподіл ролей між учасниками процесу. Власник продукту відповідає за формування бачення продукту, визначення вимог і їх пріоритезацію. Команда розробки безпосередньо виконує роботи з реалізації функціональності. Scrum-майстер забезпечує організацію процесу, усунення перешкод і дотримання принципів Scrum.

Центральним артефактом є беклог продукту, який містить упорядкований перелік вимог до системи. Вимоги у беклозі зазвичай представлені у вигляді користувацьких історій, що описують функціональність з точки зору користувача. Такий підхід дозволяє зосередитися на цінності продукту для кінцевого користувача.

Користувацька історія має структуроване формулювання, що відображає роль користувача, його потребу та очікуваний результат. Це дозволяє забезпечити зрозумілість вимог та їх орієнтацію на практичне використання.

Планування спринту є ключовим етапом процесу Scrum. Воно передбачає відбір задач із беклогу продукту, які будуть виконані у межах ітерації. При цьому враховуються пріоритети задач, їх складність і доступні ресурси команди.

Оцінювання складності задач здійснюється з використанням відносних оцінок, таких як story points. Це дозволяє врахувати не лише обсяг роботи, але й складність реалізації та рівень невизначеності. Важливо, що такі оцінки не є прямим відображенням часу, а використовуються для порівняння задач між собою.

Важливим результатом планування є формування беклогу спринту, який містить перелік задач, що будуть виконані командою. Беклог спринту визначає обсяг робіт і є основою для контролю виконання завдань.

Мета спринту визначає загальний результат, який планується досягти у межах ітерації. Вона повинна бути чітко сформульованою, узгодженою між учасниками команди та відповідати пріоритетам продукту.

У процесі виконання спринту команда проводить щоденні короткі зустрічі, що дозволяють координувати дії учасників, виявляти проблеми та контролювати прогрес виконання задач. Це забезпечує прозорість процесу розробки.

Наприкінці спринту проводиться демонстрація результатів роботи та аналіз процесу. Це дозволяє отримати зворотний зв’язок і визначити можливі шляхи покращення організації роботи у наступних ітераціях.

Таким чином, Scrum забезпечує гнучкість процесу розробки, дозволяє швидко реагувати на зміни вимог і забезпечує постійне вдосконалення програмного продукту.

Додатково слід зазначити, що ефективність застосування Scrum значною мірою залежить від узгодженості дій команди та чіткого розуміння ролей кожного учасника. Невизначеність у розподілі відповідальності або недостатня комунікація можуть призвести до зниження ефективності процесу розробки.

Крім того, важливим аспектом є підтримання актуальності беклогу продукту. Власник продукту повинен регулярно переглядати та уточнювати вимоги, що дозволяє забезпечити відповідність продукту актуальним потребам користувачів.

Використання Scrum також передбачає постійний аналіз результатів роботи команди. Це дозволяє виявляти вузькі місця у процесі розробки та впроваджувати зміни, спрямовані на підвищення ефективності роботи.

Таким чином, Scrum є не лише набором правил, а комплексним підходом до організації розробки програмного забезпечення, що базується на принципах гнучкості, прозорості та постійного вдосконалення.

Практичне завдання

1. Ознайомитися з основними принципами підходу Scrum.
2. Визначити ролі учасників процесу розробки.
3. Обрати приклад програмного продукту.
4. Сформувати початковий беклог продукту.
5. Описати вимоги у вигляді користувацьких історій.
6. Визначити пріоритети задач у беклозі.
7. Виконати оцінювання складності задач.
8. Обрати задачі для виконання у межах спринту.
9. Сформувати беклог спринту.

10. Визначити мету спринту.
11. Описати план виконання ітерації.
12. Зробити висновки щодо процесу планування.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно детально ознайомитися з концепцією гнучкої розробки програмного забезпечення та місцем Scrum серед інших підходів. Доцільно звернути увагу на принципи Agile, які лежать в основі Scrum, зокрема орієнтацію на взаємодію учасників, швидке отримання результатів і готовність до змін.

На першому етапі необхідно визначити ролі учасників процесу. Слід чітко розуміти, що власник продукту відповідає за визначення вимог і їх пріоритезацію, команда розробки – за реалізацію функціональності, а Scrum-майстер – за організацію процесу та усунення перешкод. Розуміння ролей дозволяє уникнути дублювання функцій і підвищує ефективність взаємодії.

Далі необхідно обрати програмний продукт для планування. Доцільно обирати систему, яка має декілька функціональних підсистем або ролей користувачів, що дозволить сформувати різноманітний беклог продукту.

На наступному етапі необхідно сформувати беклог продукту. Для цього потрібно визначити основні функціональні можливості системи та описати їх у вигляді користувацьких історій. Формулювання історій повинно бути чітким, орієнтованим на користувача та містити опис очікуваного результату.

Після цього необхідно виконати пріоритезацію задач. При визначенні пріоритетів слід враховувати значущість функціональності для користувача, складність реалізації та залежності між задачами. Пріоритезація дозволяє визначити послідовність виконання робіт.

Далі виконується оцінювання складності задач. Доцільно використовувати відносні оцінки, що дозволяють порівнювати задачі між собою. Оцінювання повинно виконуватися колективно, із залученням усіх членів команди, що забезпечує більш об’єктивний результат.

На основі сформованого беклогу необхідно обрати задачі для виконання у межах спринту. При цьому слід враховувати обмеження за часом, можливості команди та ризики, пов’язані з реалізацією задач.

Після цього формується беклог спринту. Важливо забезпечити, щоб обсяг робіт відповідав можливостям команди та був реалістичним для виконання у встановлений термін.

Особливу увагу необхідно приділити визначенню мети спринту. Мета повинна бути конкретною, зрозумілою та відповідати пріоритетам проєкту. Вона є орієнтиром для команди протягом усього спринту.

У процесі планування доцільно також визначити можливі ризики та шляхи їх мінімізації. Це дозволяє підвищити стабільність виконання спринту та уникнути затримок.

Після завершення планування необхідно проаналізувати сформований план. Слід оцінити його повноту, узгодженість і реалістичність. У разі виявлення перевантаження команди або недостатнього обсягу робіт необхідно внести відповідні коригування.

Таким чином, виконання роботи дозволяє сформувати практичні навички планування ітерацій розробки за підходом Scrum, організації командної роботи та управління вимогами у процесі створення програмного забезпечення.

Додатково доцільно звернути увагу на документування результатів планування. Усі сформовані артефакти, такі як беклог продукту та беклог спринту, повинні бути оформлені у зрозумілому вигляді, що дозволяє використовувати їх у подальшій роботі.

Крім того, важливо забезпечити прозорість процесу планування. Усі учасники команди повинні мати доступ до інформації про обсяг робіт, їх пріоритети та стан виконання, що сприяє підвищенню ефективності взаємодії.

Таким чином, детальне опрацювання кожного етапу планування дозволяє забезпечити ефективну організацію процесу розробки та досягнення поставлених цілей у межах спринту.

Контрольні питання

1. Що таке Scrum?
2. Які основні ролі визначено у Scrum?
3. Що таке беклог продукту?
4. Що таке користувацька історія?
5. Як виконується пріоритезація задач?
6. Що таке story points?
7. Яка мета планування ітерації?
8. Що таке беклог спринту?
9. Яку роль відіграє мета спринту?
10. Які переваги використання Scrum?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис обраного програмного продукту;
– беклог продукту;
– перелік користувацьких історій;
– оцінювання задач;
– беклог спринту;
– мета спринту;
– план ітерації;
– висновки.

ТЕМА 2. ПРОЄКТУВАННЯ ТА АРХІТЕКТУРА ПРОГРАМНИХ СИСТЕМ

Практична робота 5. Розробка UML-діаграми варіантів використання та діаграми класів

Мета роботи – ознайомитися з основами моделювання програмних систем за допомогою UML, навчитися будувати діаграми варіантів використання та діаграми класів для опису структури і поведінки програмної системи.

Ключові положення

Моделювання програмного забезпечення є важливим етапом процесу розробки, що дозволяє формалізувати вимоги, визначити структуру системи та забезпечити узгодженість між учасниками проєкту. Для цього широко використовується мова UML, яка є стандартом для візуального моделювання програмних систем.

UML надає набір діаграм, що дозволяють описувати різні аспекти системи. Серед них особливе значення мають діаграми варіантів використання та діаграми класів, які дозволяють відобразити функціональність системи та її структурну організацію.

Діаграма варіантів використання призначена для опису взаємодії користувачів із системою. Вона відображає функціональні можливості системи у вигляді варіантів використання та визначає ролі користувачів, які взаємодіють із системою. Такий підхід дозволяє сформувати загальне уявлення про функціональність програмного продукту.

Основними елементами діаграми варіантів використання є актори, варіанти використання та зв’язки між ними. Актори представляють зовнішні сутності, що взаємодіють із системою, наприклад користувачів або інші системи. Варіанти використання описують функції, які система повинна виконувати.

Зв’язки між елементами діаграми можуть відображати взаємодію, включення або розширення варіантів використання. Це дозволяє деталізувати поведінку системи та визначити залежності між різними функціями.

Діаграма класів використовується для опису статичної структури програмної системи. Вона відображає класи, їх атрибути та методи, а також зв’язки між класами. Така діаграма є основою для подальшої реалізації програмного забезпечення.

Класи є основними елементами діаграми класів. Вони визначають структуру об’єктів та їх поведінку. Атрибути описують властивості класів, а методи – операції, що виконуються над даними.

Зв’язки між класами можуть мати різний характер, зокрема асоціацію, агрегацію, композицію та успадкування. Кожен тип зв’язку відображає певний тип взаємодії між класами та має важливе значення для проєктування системи.

Успадкування дозволяє створювати нові класи на основі існуючих, що сприяє повторному використанню коду. Агрегація та композиція відображають відношення включення між класами, що дозволяє моделювати складні структури об’єктів.

Правильна побудова діаграм UML дозволяє зменшити складність системи, забезпечити її зрозумілість та спростити процес розробки. Візуальне представлення структури системи сприяє ефективній комунікації між учасниками проєкту.

Діаграми варіантів використання зазвичай створюються на ранніх етапах розробки та використовуються для уточнення вимог до системи. Діаграми класів формуються на етапі проєктування та слугують основою для реалізації програмного забезпечення.

Важливим аспектом є узгодженість між різними діаграмами. Вимоги, визначені у діаграмах варіантів використання, повинні знаходити відображення у структурі класів. Це дозволяє забезпечити цілісність моделі системи.

Крім того, використання UML дозволяє стандартизувати підходи до моделювання, що є особливо важливим у командній розробці. Єдині правила побудови діаграм забезпечують зрозумілість документації для всіх учасників проєкту.

Таким чином, UML є ефективним інструментом для опису програмних систем, що дозволяє формалізувати вимоги, визначити структуру системи та забезпечити якісну реалізацію програмного продукту.

Додатково слід зазначити, що використання діаграм дозволяє виконувати попередній аналіз архітектури системи ще до початку програмної реалізації. Це дає змогу виявити потенційні проблеми, пов’язані зі структурою або взаємодією компонентів, і внести необхідні зміни на ранніх етапах.

Також UML-діаграми можуть використовуватися як частина технічної документації програмного забезпечення. Вони забезпечують наочне представлення системи та полегшують її супроводження і подальший розвиток.

Таким чином, застосування UML сприяє підвищенню якості проєктування програмного забезпечення та ефективності командної роботи.

Практичне завдання

1. Ознайомитися з основами мови UML.
2. Обрати приклад програмної системи.
3. Визначити основних акторів системи.
4. Визначити варіанти використання системи.
5. Побудувати діаграму варіантів використання.
6. Проаналізувати взаємодію між акторами та системою.
7. Визначити основні класи системи.
8. Визначити атрибути та методи класів.
9. Визначити зв’язки між класами.
10. Побудувати діаграму класів.
11. Перевірити узгодженість між діаграмами.
12. Зробити висновки щодо побудованих моделей.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з основами UML та принципами моделювання програмних систем. Доцільно звернути увагу на призначення різних типів діаграм та їх роль у процесі розробки.

На першому етапі необхідно обрати програмну систему для моделювання. Система повинна мати достатню функціональність, щоб забезпечити побудову змістовних діаграм.

Далі необхідно визначити акторів системи. Слід враховувати не лише кінцевих користувачів, але й інші системи або зовнішні компоненти, що взаємодіють із програмним продуктом.

Після цього необхідно визначити варіанти використання. Для цього слід проаналізувати функціональні вимоги до системи та виділити основні сценарії взаємодії користувачів із системою.

На основі отриманих даних необхідно побудувати діаграму варіантів використання. Важливо забезпечити правильне відображення зв'язків між акторами та варіантами використання, а також коректно використовувати відношення включення та розширення.

Далі необхідно перейти до побудови діаграми класів. Для цього слід визначити основні сутності системи, які будуть представлені у вигляді класів.

На наступному етапі необхідно визначити атрибути та методи класів. Атрибути повинні відображати стан об'єктів, а методи – їх поведінку. Важливо забезпечити логічну відповідність між атрибутами та методами.

Після цього необхідно визначити зв'язки між класами. Слід проаналізувати взаємодію між сутностями системи та визначити типи зв'язків, що найкраще відображають їх взаємозв'язки.

У процесі побудови діаграми класів необхідно звернути увагу на узгодженість моделі. Класи повинні відповідати функціональності, визначеній у діаграмі варіантів використання.

Особливу увагу слід приділити перевірці побудованих діаграм. Необхідно переконатися, що вони є повними, несуперечливими та відповідають вимогам до системи.

Після завершення побудови діаграм доцільно провести їх аналіз. Необхідно оцінити зрозумілість моделі, її відповідність вимогам та можливість використання у процесі розробки.

Додатково слід звернути увагу на використання інструментальних засобів для побудови UML-діаграм. Це можуть бути спеціалізовані програмні продукти або універсальні графічні редактори, що підтримують створення структурованих схем.

Таким чином, виконання роботи дозволяє сформувати навички моделювання програмних систем та використання UML для опису структури і поведінки програмного забезпечення.

Крім того, доцільно забезпечити документування отриманих результатів. Побудовані діаграми повинні бути оформлені у вигляді, придатному для подальшого використання у процесі розробки та супроводження програмного продукту.

Таким чином, детальне опрацювання кожного етапу моделювання дозволяє підвищити якість проєктування програмного забезпечення та забезпечити узгодженість між різними аспектами системи.

Контрольні питання

1. Що таке UML?
2. Яке призначення діаграми варіантів використання?
3. Хто такі актори у UML?
4. Що таке варіант використання?
5. Які зв'язки використовуються у діаграмі варіантів використання?
6. Що таке діаграма класів?
7. Які елементи містить клас?

8. Які типи зв’язків існують між класами?
9. Що таке успадкування?
10. Чому важлива узгодженість між діаграмами?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис обраної програмної системи;
– діаграма варіантів використання;
– діаграма класів;
– аналіз побудованих діаграм;
– висновки.

Практична робота 6. Проєктування архітектури програмної системи та вибір архітектурного стилю

Мета роботи – ознайомитися з принципами проєктування архітектури програмного забезпечення, навчитися визначати структуру системи та обирати архітектурний стиль відповідно до вимог і характеристик програмного продукту.

Ключові положення

Архітектура програмного забезпечення визначає загальну структуру системи, її основні компоненти, способи взаємодії між ними та принципи організації програмного коду. Вона є основою для подальшої реалізації, забезпечує масштабованість, надійність і зручність супроводження програмного продукту.

Проєктування архітектури виконується на основі вимог до системи. Функціональні вимоги визначають, які задачі повинна виконувати система, тоді як нефункціональні вимоги визначають обмеження та характеристики якості, що впливають на вибір архітектурних рішень.

Одним із ключових аспектів архітектури є декомпозиція системи на компоненти. Компонентний підхід дозволяє розділити систему на окремі частини, кожна з яких виконує визначену функцію. Це сприяє зменшенню складності системи та полегшує її розробку і супроводження.

Важливим принципом є слабке зв’язування компонентів і висока зв’язність внутрішньої структури. Це означає, що компоненти повинні бути максимально незалежними один від одного, а їх внутрішня структура повинна бути логічно узгодженою.

Існує декілька архітектурних стилів, що використовуються при розробці програмного забезпечення. Найбільш поширеними є шарова архітектура, клієнт-серверна архітектура, мікросервісна архітектура та подієво-орієнтована архітектура.

Шарова архітектура передбачає поділ системи на рівні, кожен із яких виконує визначені функції. Наприклад, можуть виділятися рівні представлення, бізнес-логіки та доступу до даних. Такий підхід забезпечує зрозумілу структуру системи.

Клієнт-серверна архітектура базується на взаємодії між клієнтською частиною, що формує запити, та серверною частиною, що їх обробляє. Такий підхід широко використовується у вебзастосунках.

Мікросервісна архітектура передбачає побудову системи як набору незалежних сервісів, що взаємодіють між собою через стандартизовані інтерфейси. Це забезпечує гнучкість та можливість масштабування системи.

Подієво-орієнтована архітектура базується на обробці подій, що виникають у системі. Компоненти реагують на події, що дозволяє забезпечити асинхронну взаємодію та підвищити масштабованість.

Вибір архітектурного стилю залежить від типу системи, вимог до продуктивності, масштабованості, надійності та інших характеристик. Неправильний вибір архітектури може призвести до складності реалізації та обмежень у розвитку системи.

Проєктування архітектури також передбачає визначення інтерфейсів взаємодії між компонентами. Чітко визначені інтерфейси дозволяють забезпечити незалежність компонентів та спростити інтеграцію системи.

Важливим аспектом є документування архітектурних рішень. Архітектурна документація дозволяє забезпечити зрозумілість структури системи та полегшує її супроводження.

Таким чином, проєктування архітектури програмного забезпечення є ключовим етапом розробки, що визначає ефективність створення та подальшого використання програмного продукту.

Додатково слід зазначити, що архітектура програмної системи повинна враховувати можливість подальшого розширення. Це означає, що при проєктуванні необхідно передбачати можливість додавання нових компонентів без значної зміни існуючої структури.

Також важливо враховувати вплив обраної архітектури на продуктивність системи. Наприклад, використання розподілених архітектур може підвищити масштабованість, але водночас ускладнює управління взаємодією між компонентами.

Таким чином, вибір архітектурного стилю є компромісом між різними вимогами та обмеженнями, що висуваються до програмної системи.

Практичне завдання

1. Ознайомитися з поняттям архітектури програмного забезпечення.
2. Проаналізувати вимоги до програмної системи.
3. Визначити основні компоненти системи.
4. Виконати декомпозицію системи на складові частини.
5. Ознайомитися з основними архітектурними стилями.
6. Обрати архітектурний стиль для реалізації системи.
7. Обґрунтувати вибір архітектурного стилю.
8. Визначити інтерфейси взаємодії між компонентами.
9. Побудувати схему архітектури системи.
10. Проаналізувати переваги та недоліки обраної архітектури.
11. Зробити висновки щодо ефективності обраного рішення.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з основами проєктування архітектури програмного забезпечення. Доцільно звернути увагу на принципи декомпозиції системи та організації взаємодії між її компонентами.

На першому етапі необхідно проаналізувати вимоги до системи. Слід визначити функціональні можливості та нефункціональні характеристики, які будуть впливати на вибір архітектури.

Далі необхідно визначити основні компоненти системи. Для цього слід виділити логічні частини системи, кожна з яких виконує визначені функції.

Після цього необхідно виконати декомпозицію системи. Слід розділити систему на компоненти таким чином, щоб забезпечити їх незалежність та зручність взаємодії.

На наступному етапі необхідно ознайомитися з різними архітектурними стилями та визначити їх особливості. Це дозволить обґрунтовано обрати найбільш відповідний підхід.

Далі необхідно обрати архітектурний стиль для реалізації системи. Вибір повинен базуватися на аналізі вимог та характеристик системи.

Після цього необхідно визначити інтерфейси взаємодії між компонентами. Слід описати способи обміну даними та взаємодії між різними частинами системи.

На основі отриманих даних необхідно побудувати схему архітектури системи. Вона повинна відображати компоненти системи та їх взаємозв’язки.

У процесі виконання роботи необхідно звернути увагу на узгодженість архітектурних рішень із вимогами до системи. Архітектура повинна забезпечувати реалізацію всіх функціональних можливостей.

Особливу увагу слід приділити аналізу переваг і недоліків обраного архітектурного стилю. Це дозволяє оцінити доцільність прийнятого рішення.

Після завершення роботи необхідно зробити висновки щодо ефективності обраної архітектури та можливості її застосування у подібних проєктах.

Додатково доцільно звернути увагу на документування архітектурних рішень. Опис архітектури повинен бути зрозумілим і містити всі необхідні відомості для подальшої реалізації системи.

Крім того, важливо забезпечити можливість подальшого розвитку системи. Архітектура повинна бути гнучкою та адаптивною до змін вимог.

Таким чином, детальне опрацювання архітектури дозволяє забезпечити ефективну реалізацію програмного продукту та його подальший розвиток.

Контрольні питання

1. Що таке архітектура програмного забезпечення?
2. Які фактори впливають на вибір архітектури?
3. Що таке декомпозиція системи?
4. Які існують архітектурні стилі?
5. У чому полягає сутність шарової архітектури?
6. Які особливості клієнт-серверної архітектури?
7. Що таке мікросервісна архітектура?
8. У чому полягає подієво-орієнтований підхід?
9. Чому важливо визначати інтерфейси між компонентами?
10. Яку роль відіграє архітектурна документація?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис програмної системи;
– перелік вимог;
– опис компонентів системи;
– обраний архітектурний стиль;
– схема архітектури;
– аналіз переваг і недоліків;
– висновки.

Практична робота 7. Декомпозиція системи на модулі та визначення інтерфейсів взаємодії

Мета роботи – ознайомитися з принципами модульної побудови програмних систем, навчитися виконувати декомпозицію системи на окремі модулі та визначати інтерфейси взаємодії між ними відповідно до функціональних вимог і архітектурних рішень.

Ключові положення

Декомпозиція програмної системи є етапом проєктування, що полягає у поділі складної системи на окремі частини з визначеними функціями. Це дозволяє зменшити складність розробки, спростити аналіз структури та забезпечити паралельну роботу над компонентами. У сучасній інженерії програмного забезпечення декомпозиція розглядається як засіб підвищення керованості проєкту.

Модуль є логічно завершеною частиною системи, що реалізує певну підмножину функцій і взаємодіє з іншими через інтерфейси. Виділення модулів базується на функціональних вимогах, сценаріях використання та архітектурних рішеннях. Основними принципами є висока внутрішня зв’язність і слабкий зв’язок між модулями, що підвищує гнучкість, спрощує тестування та зменшує вплив змін.

Декомпозиція може виконуватися за функціональним підходом, за предметними сутностями або за рівнями відповідно до шарової архітектури. Вона повинна враховувати як функціональні, так і нефункціональні вимоги, зокрема продуктивність, безпеку та масштабованість. Правильний поділ системи дозволяє створювати незалежні компоненти, що можуть розвиватися окремо.

Важливим елементом є визначення інтерфейсів взаємодії, які формалізують обмін даними та виклики функцій між модулями. Інтерфейси можуть бути внутрішніми або зовнішніми та повинні містити чіткий опис операцій, параметрів, результатів і можливих помилок. Це забезпечує стабільність взаємодії та ізоляцію внутрішньої реалізації.

Модульна організація передбачає приховування реалізації, що відповідає принципу інкапсуляції та дозволяє змінювати внутрішню логіку без впливу на інші частини системи. Водночас важливо уникати як надмірного укрупнення, так і надмірного дроблення модулів, забезпечуючи збалансовану структуру.

Декомпозиція сприяє модульному тестуванню, повторному використанню компонентів і спрощенню супроводження системи. Документування модулів та інтерфейсів забезпечує прозорість розробки та підтримку системи. Оскільки структура може змінюватися у процесі розробки, декомпозиція повинна бути достатньо гнучкою для коригування.

Практичне завдання

1. Ознайомитися з поняттям декомпозиції програмної системи.
2. Проаналізувати вимоги до обраної програмної системи.
3. Визначити основні функціональні частини системи.
4. Виконати поділ системи на модулі.
5. Визначити призначення кожного модуля.
6. Встановити зв’язки між модулями.
7. Визначити інтерфейси взаємодії між модулями.
8. Описати вхідні та вихідні дані для кожного інтерфейсу.
9. Побудувати схему модульної структури системи.
10. Проаналізувати доцільність обраної декомпозиції.
11. Зробити висновки щодо якості побудованої модульної структури.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з основами модульного проєктування програмних систем. Доцільно звернути увагу на принципи поділу системи на логічно завершені частини, а також на вимоги до побудови інтерфейсів взаємодії. Важливо усвідомити, що метою декомпозиції є не просто формальний поділ системи, а створення зручної та обґрунтованої структури, придатної для подальшої реалізації.

На першому етапі необхідно обрати програмну систему або використати приклад, що розглядався у попередніх роботах. Доцільно, щоб система мала достатню кількість функціональних можливостей, оскільки це дозволить побудувати змістовну модульну структуру. Перед початком декомпозиції слід коротко описати призначення системи, її основних користувачів та головні функції.

Далі необхідно проаналізувати вимоги до системи. Слід визначити, які функції повинні бути реалізовані, які дані обробляються, які обмеження існують щодо продуктивності, безпеки або масштабованості. Саме вимоги повинні стати основою для поділу системи на модулі. Якщо декомпозиція не спирається на вимоги, вона може виявитися випадковою та неефективною.

На наступному етапі потрібно виділити основні функціональні частини системи. Наприклад, у прикладній інформаційній системі окремо можуть бути виділені модулі реєстрації користувачів, обробки замовлень, керування даними, формування звітів, адміністрування та інтеграції з іншими сервісами. Для кожного такого модуля необхідно визначити його призначення та межі відповідальності.

Після цього необхідно виконати безпосередню декомпозицію. Поділ системи слід здійснювати так, щоб кожен модуль відповідав за окрему групу функцій і не дублював призначення інших модулів. Доцільно уникати ситуації, коли одна й та сама функція частково реалізується у кількох модулях без чітких меж. Також не слід створювати модулі, призначення яких є надто широким або, навпаки, надто вузьким.

Під час визначення модулів необхідно для кожного з них сформулювати короткий опис. У цьому описі доцільно зазначити, які функції реалізує модуль, з якими даними працює, які операції виконує та з якими іншими частинами системи взаємодіє. Такий опис дозволяє перевірити, чи справді виділений модуль є логічно цілісним.

Далі необхідно встановити зв’язки між модулями. Для цього слід проаналізувати, які модулі обмінюються даними, які з них викликають функції інших модулів, які компоненти є залежними від результатів роботи інших частин системи. Важливо не лише зафіксувати факт взаємодії, але й визначити її напрям і характер. Наприклад, один модуль може передавати дані для обробки, інший – повертати результат, а третій – надавати сервісні функції.

На наступному етапі потрібно визначити інтерфейси взаємодії між модулями. Для кожного інтерфейсу доцільно встановити, які саме операції або сервіси він надає, які дані приймає, які результати повертає та за яких умов виконується взаємодія. Якщо взаємодія пов’язана з передаванням структурованих даних, слід коротко описати склад цих даних або їх формат.

Особливу увагу необхідно приділити чіткості опису інтерфейсів. Формулювання повинні бути однозначними та придатними для практичного використання під час реалізації системи. Наприклад, недостатньо вказати, що один модуль передає інформацію іншому. Потрібно уточнити, яка саме інформація передається, у якому вигляді, за яких умов і який результат очікується у відповідь.

Після визначення інтерфейсів доцільно побудувати схему модульної структури системи. На схемі потрібно відобразити всі основні модулі та зв’язки між ними. Якщо це можливо, варто окремо позначити напрями взаємодії. Така схема дозволяє візуально оцінити структуру системи, виявити зайві або недостатньо обґрунтовані зв’язки та перевірити логічність побудови.

У процесі виконання роботи слід проаналізувати, наскільки обрана декомпозиція відповідає вимогам до системи. Потрібно оцінити, чи не перевантажені окремі модулі надмірною кількістю функцій, чи немає дублювання відповідальності, чи забезпечено зручну взаємодію між частинами системи. Якщо виявлено структурні недоліки, доцільно скоригувати поділ системи ще на етапі проєктування.

Також важливо оцінити, наскільки побудована модульна структура сприяє подальшій реалізації та тестуванню системи. Правильна декомпозиція повинна полегшувати розподіл задач між розробниками, спрощувати локалізацію помилок та забезпечувати можливість незалежної перевірки окремих модулів. Якщо модулі є надто залежними один від одного, це свідчить про недостатню якість декомпозиції.

Додатково доцільно звернути увагу на перспективу подальшого розвитку системи. Слід оцінити, чи дозволяє побудована структура додавати нові функції без радикальної зміни вже визначених модулів. Це особливо важливо для систем, що будуть розширюватися або інтегруватися з іншими програмними продуктами.

Після завершення роботи необхідно сформулювати висновки. У висновках слід коротко охарактеризувати отриману модульну структуру, оцінити якість визначених інтерфейсів взаємодії та вказати, які переваги забезпечує обрана декомпозиція для подальшої розробки програмної системи.

Таким чином, послідовне виконання всіх етапів роботи дозволяє сформувати практичні навички декомпозиції програмної системи, визначення меж відповідальності модулів та побудови інтерфейсів взаємодії, що є важливою складовою проєктування програмного забезпечення.

Контрольні питання

1. Що таке декомпозиція програмної системи?
2. Що таке модуль у програмному забезпеченні?
3. Які принципи слід враховувати під час декомпозиції системи?
4. Що означає висока внутрішня зв’язність модуля?
5. Що означає слабкий зв’язок між модулями?
6. Що таке інтерфейс взаємодії між модулями?
7. Які дані доцільно вказувати в описі інтерфейсу?
8. Чому важливо уникати дублювання функцій у модулях?
9. Яким чином модульна структура впливає на тестування системи?
10. Чому документування модулів та інтерфейсів є важливим?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис програмної системи;
– перелік модулів системи;
– характеристика призначення кожного модуля;
– опис інтерфейсів взаємодії;
– схема модульної структури системи;
– аналіз доцільності обраної декомпозиції;
– висновки.

Практична робота 8. Реалізація прототипу компонента з визначеним API

Мета роботи – ознайомитися з принципами реалізації програмних компонентів із чітко визначеним прикладним програмним інтерфейсом, навчитися створювати прототип компонента та перевіряти коректність його взаємодії з іншими частинами системи.

Ключові положення

У сучасній розробці програмного забезпечення важливу роль відіграє компонентний підхід, за якого система розглядається як сукупність взаємодіючих частин із визначеними інтерфейсами. Це забезпечує структурованість, спрощує розподіл задач і підвищує керованість розробки. Практичною реалізацією такого підходу є створення компонентів із визначеним API.

API є інтерфейсом, через який компонент надає свої функції іншим частинам системи. Він визначає операції, параметри, формати даних і способи обробки помилок, що дозволяє відокремити реалізацію від використання. Прототип компонента є спрощеною реалізацією, яка дає змогу перевірити архітектурні рішення, інтерфейс і можливість інтеграції на ранніх етапах.

Під час розробки прототипу важливо чітко розмежовувати інтерфейс і реалізацію. API повинен бути зрозумілим, узгодженим із вимогами та однозначним, тоді як внутрішня

реалізація може бути спрощеною. Компонент реалізує конкретний набір операцій, що описуються через параметри, результати та умови виникнення помилок.

Значну увагу слід приділяти обробці помилок і зручності використання API. Прототип дозволяє виявити недоліки інтерфейсу та уточнити його структуру. Документування API є обов'язковим і повинно містити опис операцій, параметрів і результатів.

API виступає як контракт взаємодії між компонентами, що дозволяє організувати паралельну розробку. Прототипування дає змогу перевірити технічну здійсненність рішень і підготувати основу для подальшої реалізації системи.

Практичне завдання

Ознайомитися з поняттям програмного компонента та API.
Обрати компонент програмної системи для реалізації.
Визначити призначення компонента та його місце в архітектурі системи.
Сформувати перелік операцій, що повинні надаватися через API.
Визначити вхідні та вихідні дані для кожної операції.
Визначити можливі помилки та способи їх обробки.
Реалізувати прототип компонента.
Підготувати приклади виклику операцій API.
Перевірити коректність роботи реалізованого прототипу.
Проаналізувати зручність і повноту реалізованого API.
Зробити висновки щодо придатності компонента до подальшої інтеграції.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з принципами компонентного проєктування та роллю API у програмних системах. Доцільно звернути увагу на те, що реалізація компонента повинна починатися не з написання коду, а з осмислення його призначення, меж відповідальності та способів взаємодії з іншими частинами системи. Саме чітке визначення API дозволяє забезпечити зрозумілість, повторне використання та можливість інтеграції компонента.

На першому етапі необхідно обрати компонент для реалізації. Доцільно використовувати один із модулів, визначених у попередній роботі. Це дозволить забезпечити логічний зв'язок між роботами та продовжити розробку програмної системи на основі вже сформованої модульної структури. Обраний компонент повинен мати достатньо чітке призначення і виконувати конкретну групу функцій.

Далі необхідно описати призначення компонента. У цьому описі слід зазначити, які задачі він вирішує, які дані обробляє, з якими іншими компонентами взаємодіє та яке місце займає у загальній архітектурі системи. Такий попередній аналіз дозволяє уникнути помилок, пов'язаних із надмірно широким або нечітким трактуванням функцій компонента.

Після цього потрібно визначити API компонента. Для цього слід сформувати перелік операцій, які будуть доступні зовнішнім модулям або користувачам системи. Назви операцій повинні бути змістовними та відображати їх реальне призначення. Для кожної операції необхідно визначити набір параметрів, типи вхідних даних, очікуваний результат і можливі повідомлення про помилки. Якщо компонент працює з об'єктами певного типу, доцільно коротко описати структуру цих об'єктів.

На наступному етапі слід проаналізувати можливі сценарії використання API. Необхідно визначити, у яких ситуаціях буде викликатися кожна операція, які дані надходитимуть на вхід, який результат очікується та які виняткові ситуації можуть виникати. Це дозволяє переконатися, що інтерфейс є не лише формально повним, але й придатним для практичного використання.

Далі необхідно виконати реалізацію прототипу компонента. Реалізація може бути спрощеною, однак вона повинна підтримувати основні операції, визначені в API. Якщо повноцінна робота з реальними джерелами даних є недоцільною, допускається використання тестових або умовних даних. Важливо, щоб реалізація дозволяла перевірити логіку виклику операцій та правильність обробки результатів.

Під час реалізації компонента слід приділити увагу структурі програмного коду. Доцільно організувати код таким чином, щоб зовнішні операції API були чітко відокремлені від внутрішніх допоміжних процедур. Це відповідає принципу приховування реалізації та дозволяє легше змінювати внутрішню логіку компонента без зміни зовнішнього інтерфейсу.

Особливу увагу необхідно приділити обробці помилок. Для кожної операції потрібно передбачити ситуації некоректного введення, відсутності необхідних даних або інших умов, за яких операція не може бути виконана успішно. У межах прототипу помилки можуть оброблятися у спрощеному вигляді, однак вони повинні бути враховані та відображені у результатах роботи компонента.

Після реалізації прототипу необхідно підготувати приклади використання API. Це можуть бути окремі фрагменти коду, тестові виклики методів або приклади запитів і відповідей. Такі приклади дозволяють перевірити зручність використання компонента та продемонструвати, як саме він взаємодіє з іншими частинами системи.

На наступному етапі потрібно перевірити коректність роботи реалізованого прототипу. Для цього слід виконати тестові звернення до всіх основних операцій API, перевірити правильність повернених результатів і проаналізувати поведінку компонента у разі помилкових вхідних даних. У разі виявлення недоліків доцільно скоригувати інтерфейс або реалізацію.

Також необхідно оцінити зручність і повноту розробленого API. Потрібно проаналізувати, чи всі необхідні операції включено до інтерфейсу, чи не містить API зайвих або дубльованих функцій, чи є зрозумілими назви операцій і параметрів, чи зручно використовувати компонент з точки зору іншого розробника або користувача системи.

Додатково доцільно виконати коротке документування створеного API. Для кожної операції бажано подати стислий опис призначення, вхідних параметрів, результату та можливих помилок. Якщо це передбачено форматом роботи, таку інформацію можна подати у вигляді таблиці або структурованого переліку. Документування сприяє впорядкованості результатів і полегшує подальше використання компонента.

Після завершення роботи необхідно сформулювати висновки. У висновках слід оцінити, наскільки створений прототип відповідає визначеним вимогам, чи є його API достатньо повним і зрозумілим, а також чи може цей компонент бути використаний як основа для подальшої інтеграції у програмну систему.

Таким чином, послідовне виконання роботи дозволяє сформувати практичні навички проєктування і реалізації програмних компонентів, визначення прикладних програмних інтерфейсів та перевірки їх придатності для використання у складі більш складних програмних систем.

Контрольні питання

1. Що таке програмний компонент?
2. Що таке API?
3. Яке призначення API у програмній системі?
4. Чим відрізняється зовнішній інтерфейс компонента від його внутрішньої реалізації?
5. Що таке прототип компонента?
6. Які елементи повинні бути визначені для кожної операції API?
7. Чому важливо враховувати обробку помилок під час побудови API?
8. Яку роль відіграє документування API?
9. Чому прототип компонента доцільно перевіряти на тестових сценаріях?
10. Яким чином реалізація прототипу сприяє подальшій інтеграції системи?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис обраного компонента;
– місце компонента в архітектурі системи;
– опис API компонента;
– приклади виклику операцій;
– результати перевірки роботи прототипу;
– аналіз повноти та зручності API;
– висновки.

Практична робота 9. Інтеграція декількох модулів у єдину систему

Мета роботи – ознайомитися з принципами інтеграції програмних компонентів, навчитися поєднувати окремі модулі у цілісну систему та перевіряти коректність їх взаємодії.

Ключові положення

Інтеграція модулів є одним із ключових етапів розробки програмного забезпечення, на якому окремі компоненти системи об’єднуються в єдину функціональну структуру. Якщо на попередніх етапах розробки основна увага приділяється окремим модулям і їх реалізації, то на етапі інтеграції важливим стає забезпечення коректної взаємодії між ними та узгодженості всієї системи.

Модулі, що були розроблені незалежно, повинні взаємодіяти через визначені інтерфейси. Саме ці інтерфейси виступають основою інтеграції. Якщо інтерфейси описані чітко і реалізовані відповідно до специфікації, процес інтеграції значно спрощується. У протилежному випадку виникають проблеми сумісності, що потребують додаткових змін у коді.

Однією з основних задач інтеграції є узгодження форматів даних, що передаються між модулями. Дані, які формує один модуль, повинні бути коректно інтерпретовані іншим модулем. Це вимагає єдиного підходу до представлення даних, їх структури та типів. У разі

невідповідності форматів необхідно реалізовувати механізми перетворення даних або коригувати інтерфейси.

Інтеграція може виконуватися за різними стратегіями. Одним із підходів є поетапна інтеграція, коли модулі об'єднуються поступово, перевіряючи коректність взаємодії на кожному кроці. Іншим підходом є одночасна інтеграція всіх модулів, що є швидшою, але значно складнішою з точки зору виявлення помилок. У практиці розробки частіше застосовується поступовий підхід, оскільки він дозволяє локалізувати проблеми на ранніх етапах.

Під час інтеграції важливу роль відіграє тестування. Інтеграційне тестування спрямоване на перевірку взаємодії між модулями та виявлення помилок, пов'язаних із передаванням даних, викликом функцій або узгодженням логіки роботи. Таке тестування дозволяє переконатися, що система функціонує як єдине ціле.

Особливу увагу необхідно приділяти порядку виклику модулів. У складних системах один модуль може залежати від результатів роботи іншого. Тому необхідно забезпечити правильну послідовність виконання операцій, що гарантує коректність роботи всієї системи.

Інтеграція також пов'язана з питанням керування залежностями між модулями. Якщо залежності є надто складними або циклічними, це ускладнює процес інтеграції та супроводження системи. Тому на етапі проєктування доцільно мінімізувати залежності та забезпечити їх зрозумілу структуру.

У процесі інтеграції часто використовуються допоміжні механізми, такі як заглушки та імітатори. Вони дозволяють перевіряти взаємодію модулів навіть у випадку, коли деякі частини системи ще не реалізовані. Це сприяє паралельній розробці та скороченню часу створення програмного продукту.

Важливим аспектом є обробка помилок під час взаємодії модулів. У разі виникнення помилки в одному модулі система повинна коректно реагувати, не допускаючи порушення роботи інших частин. Це вимагає узгодженого підходу до обробки виняткових ситуацій.

Інтеграція модулів безпосередньо пов'язана з архітектурою системи. Вибраний архітектурний стиль визначає спосіб взаємодії між компонентами, характер зв'язків і механізми обміну даними. Наприклад, у шаровій архітектурі взаємодія відбувається між рівнями, тоді як у мікросервісній архітектурі – через мережеві інтерфейси.

Додатково слід зазначити, що інтеграція є не одноразовою дією, а процесом, що може повторюватися на різних етапах розробки. У міру додавання нових функцій або модулів система потребує повторної інтеграції та перевірки взаємодії між компонентами.

Також важливо враховувати питання продуктивності під час інтеграції. Взаємодія між модулями може впливати на швидкість роботи системи, особливо якщо вона пов'язана з передаванням великих обсягів даних або використанням мережевих запитів.

Таким чином, інтеграція модулів є складним і відповідальним етапом розробки, що вимагає узгодженості інтерфейсів, коректності обміну даними, перевірки взаємодії та врахування архітектурних особливостей програмної системи.

Практичне завдання

1. Ознайомитися з поняттям інтеграції програмних модулів.
2. Обрати декілька модулів програмної системи для інтеграції.
3. Проаналізувати інтерфейси взаємодії між модулями.
4. Перевірити узгодженість форматів даних.

5. Реалізувати взаємодію між модулями.
6. Забезпечити правильний порядок виклику модулів.
7. Виконати інтеграцію модулів у єдину систему.
8. Провести інтеграційне тестування.
9. Виявити та усунути помилки взаємодії.
10. Проаналізувати результати інтеграції.
11. Зробити висновки щодо коректності роботи системи.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з принципами інтеграції програмних модулів та роллю інтерфейсів у забезпеченні взаємодії між компонентами. Доцільно звернути увагу на те, що інтеграція повинна базуватися на раніше визначених інтерфейсах, а не на довільному поєднанні програмного коду.

На першому етапі необхідно обрати декілька модулів, які були реалізовані у попередніх роботах. Це дозволить забезпечити логічну послідовність виконання дисципліни та сформувати єдину систему на основі вже створених компонентів. Важливо, щоб обрані модулі мали визначені точки взаємодії.

Далі необхідно проаналізувати інтерфейси взаємодії між модулями. Слід перевірити, чи відповідають вони один одному, чи узгоджені формати даних, чи правильно визначені параметри та результати викликів. У разі виявлення невідповідностей необхідно внести коригування до інтерфейсів або реалізації модулів.

На наступному етапі потрібно перевірити формати даних, що передаються між модулями. Необхідно переконатися, що структура даних є однаковою з обох сторін взаємодії, а типи даних відповідають очікуванням. У разі необхідності слід реалізувати перетворення даних або адаптаційні механізми.

Після цього необхідно реалізувати взаємодію між модулями. Це може бути виклик функцій, обмін повідомленнями або інші способи взаємодії залежно від архітектури системи. Важливо забезпечити коректну передачу даних і отримання результатів.

Особливу увагу необхідно приділити порядку виклику модулів. Слід визначити, який модуль ініціює взаємодію, які модулі виконують обробку даних і в якій послідовності відбувається виконання операцій. Неправильна організація послідовності може призвести до помилок у роботі системи.

На етапі інтеграції доцільно використовувати поетапний підхід. Спочатку інтегруються два модулі, після чого перевіряється їх взаємодія. Далі до системи додаються наступні модулі. Такий підхід дозволяє своєчасно виявляти та усувати помилки.

Після виконання інтеграції необхідно провести тестування. Слід перевірити як окремі сценарії взаємодії, так і загальну роботу системи. Особливу увагу необхідно приділити граничним випадкам і ситуаціям, що можуть викликати помилки.

У процесі тестування необхідно зафіксувати всі виявлені помилки та проаналізувати їх причини. Це можуть бути помилки у визначенні інтерфейсів, невідповідність форматів даних або неправильна логіка взаємодії. Після цього необхідно усунути виявлені недоліки.

Додатково слід звернути увагу на обробку помилок під час взаємодії модулів. Система повинна коректно реагувати на помилки та забезпечувати стабільність роботи навіть у разі некоректних вхідних даних або збоїв окремих компонентів.

Після завершення інтеграції необхідно оцінити ефективність взаємодії між модулями. Слід проаналізувати, чи відповідає система вимогам, чи забезпечено правильну передачу даних та чи є зручною структура взаємодії між компонентами.

Таким чином, виконання роботи дозволяє сформувати практичні навички інтеграції програмних модулів, перевірки їх взаємодії та забезпечення цілісності програмної системи.

Контрольні питання

1. Що таке інтеграція програмних модулів?
2. Яка роль інтерфейсів у процесі інтеграції?
3. Які існують підходи до інтеграції модулів?
4. Що таке інтеграційне тестування?
5. Чому важлива узгодженість форматів даних?
6. Які проблеми можуть виникати під час інтеграції?
7. Що таке заглушки та імітатори?
8. Чому важливо визначати порядок виклику модулів?
9. Як впливає архітектура системи на інтеграцію?
10. Яку роль відіграє обробка помилок під час інтеграції?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис використаних модулів;
– схема взаємодії модулів;
– опис реалізації інтеграції;
– результати тестування;
– аналіз виявлених помилок;
– висновки.

ТЕМА 3. ТЕХНОЛОГІЇ РЕАЛІЗАЦІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Практична робота 10. Створення репозиторію Git та організація роботи з гілками

Мета роботи – ознайомитися з принципами керування версіями програмного забезпечення, навчитися створювати репозиторій Git, організовувати структуру гілок та забезпечувати ефективну командну роботу над проєктом.

Ключові положення

Система керування версіями є важливою складовою розробки програмного забезпечення, оскільки дозволяє відстежувати зміни, зберігати історію проєкту та організовувати спільну роботу. Однією з найбільш поширених систем є Git, що реалізує розподілену модель зберігання даних і дозволяє кожному розробнику мати повну копію репозиторію разом із історією змін.

Репозиторій Git містить файли проєкту та історію їх змін і може бути локальним або віддаленим. Взаємодія між ними здійснюється через операції передавання змін. Основною одиницею фіксації є коміт, що відображає завершений набір змін і містить інформацію про автора та час. Змістовні повідомлення комітів дозволяють зрозуміти характер змін без аналізу коду.

Гілки забезпечують можливість паралельної розробки різних частин системи. Основна гілка зазвичай містить стабільну версію, а додаткові використовуються для нових функцій або виправлення помилок. Після завершення роботи зміни інтегруються через злиття, під час якого можуть виникати конфлікти, що потребують ручного вирішення.

Використання віддалених репозиторіїв забезпечує синхронізацію змін, централізоване зберігання та резервне копіювання. Важливо дотримуватися правил іменування гілок, регулярно фіксувати зміни та підтримувати актуальність репозиторію. Перед інтеграцією змін доцільно виконувати їх перевірку.

Таким чином, використання Git і організація роботи з гілками забезпечують контроль змін, узгодженість роботи команди та стабільність програмного продукту.

Практичне завдання

1. Ознайомитися з основами системи керування версіями Git.
2. Встановити Git та налаштувати середовище роботи.
3. Створити локальний репозиторій.
4. Додати файли до репозиторію.
5. Виконати перший коміт.
6. Створити віддалений репозиторій.
7. Налаштувати зв’язок між локальним і віддаленим репозиторієм.
8. Створити нову гілку.
9. Виконати зміни у гілці та зафіксувати їх.
10. Об’єднати гілки.
11. Виявити та вирішити конфлікти (за наявності).
12. Зробити висновки щодо організації роботи з Git.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з основами систем керування версіями та принципами роботи Git. Доцільно звернути увагу на поняття репозиторію, коміту, гілки та віддаленого сховища. Важливо розуміти, що Git використовується не лише для збереження файлів, а й для організації процесу розробки.

На першому етапі необхідно встановити Git і виконати базове налаштування. Слід визначити ім’я користувача та адресу електронної пошти, що будуть використовуватися для ідентифікації автора змін. Це забезпечує коректне відображення інформації у комітах.

Далі необхідно створити локальний репозиторій. Для цього слід ініціалізувати репозиторій у вибраній директорії та додати до нього файли проєкту. Після цього потрібно виконати перший коміт, що фіксує початковий стан системи.

На наступному етапі необхідно створити віддалений репозиторій та встановити зв’язок із локальним. Це дозволяє зберігати копію проєкту на сервері та забезпечує можливість спільної роботи. Після налаштування зв’язку доцільно виконати передавання змін до віддаленого репозиторію.

Далі необхідно ознайомитися з роботою з гілками. Слід створити нову гілку та виконати у ній зміни. Це може бути додавання нового функціоналу або зміна існуючого коду. Важливо зафіксувати зміни у вигляді комітів із чіткими повідомленнями.

Після завершення роботи у гілці необхідно виконати її об’єднання з основною гілкою. У разі виникнення конфліктів необхідно проаналізувати зміни та визначити правильний варіант їх об’єднання. Це вимагає уважності та розуміння внесених змін.

Особливу увагу необхідно приділити організації роботи з гілками. Доцільно використовувати окремі гілки для різних задач та забезпечувати їх своєчасне об’єднання. Це дозволяє уникнути накопичення змін і спрощує процес інтеграції.

У процесі виконання роботи слід звернути увагу на структуру історії комітів. Вона повинна бути логічною та відображати етапи розвитку проєкту. Доцільно уникати хаотичних змін і прагнути до впорядкованості історії.

Додатково слід перевірити синхронізацію між локальним і віддаленим репозиторіями. Необхідно переконатися, що всі зміни коректно передаються та відображаються у віддаленому сховищі.

Після завершення роботи необхідно проаналізувати процес використання Git. Слід оцінити зручність роботи з гілками, ефективність організації змін та можливість використання такого підходу у реальних проєктах.

Таким чином, виконання роботи дозволяє сформувати практичні навички використання системи Git, організації роботи з гілками та забезпечення ефективної взаємодії під час розробки програмного забезпечення.

Контрольні питання

1. Що таке система керування версіями?
2. Що таке Git?
3. Що таке репозиторій?
4. Що таке коміт?
5. Що таке гілка?
6. Для чого використовуються гілки?

7. Що таке злиття гілок?
8. Що таке конфлікт у Git?
9. Яка роль віддаленого репозиторію?
10. Які переваги використання Git?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис створеного репозиторію;
– структура гілок;
– приклади комітів;
– результати об’єднання гілок;
– аналіз процесу роботи;
– висновки.

Практична робота 11. Виконання злиття змін та розв’язання конфліктів у репозиторії

Мета роботи – ознайомитися з процесом об’єднання змін у системі керування версіями Git, навчитися виконувати злиття гілок та розв’язувати конфлікти, що виникають під час інтеграції змін.

Ключові положення

У процесі розробки різні частини функціональності реалізуються в окремих гілках репозиторію, що забезпечує паралельну роботу. На певному етапі виникає потреба об’єднання змін, що називається злиттям і є невід’ємною частиною роботи з системами керування версіями.

Злиття в Git передбачає об’єднання історій гілок. Якщо зміни не перетинаються, воно виконується автоматично. У разі зміни одних і тих самих ділянок коду виникає конфлікт, який потребує участі розробника. Конфлікти можуть виникати у будь-яких файлах і зазвичай спричинені паралельним редагуванням пов’язаних частин системи.

Розв’язання конфліктів полягає в аналізі варіантів змін і формуванні коректного результату. У файлах вони позначаються спеціальними маркерами, які необхідно видалити після редагування. Важливо враховувати логіку програми та взаємозв’язки між компонентами, оскільки помилки на цьому етапі можуть призвести до некоректної роботи системи.

Злиття може виконуватися різними способами, зокрема стандартним об’єднанням або перебазуванням. Перед злиттям слід оновити репозиторій і зафіксувати всі зміни, а після — перевірити працездатність системи.

Конфлікти є природною частиною розробки, а їх кількість зменшується за рахунок регулярної синхронізації, використання невеликих гілок і впорядкованої організації роботи. Таким чином, злиття змін і розв’язання конфліктів забезпечують узгодженість коду та ефективну командну взаємодію.

Практичне завдання

1. Ознайомитися з поняттям злиття змін у Git.
2. Створити дві гілки у репозиторії.
3. Виконати зміни у кожній гілці.
4. Зафіксувати зміни у вигляді комітів.
5. Виконати злиття гілок.
6. Виявити конфлікти (за наявності).
7. Проаналізувати причини виникнення конфліктів.
8. Виконати розв’язання конфліктів.
9. Завершити процес злиття.
10. Перевірити коректність об’єднаного коду.
11. Зробити висновки щодо процесу злиття.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з принципами роботи системи Git, зокрема з поняттями гілки, коміту та злиття. Важливо розуміти, що злиття є процесом об’єднання змін, які виконувалися незалежно, і тому може супроводжуватися виникненням конфліктів.

На першому етапі необхідно створити дві гілки у репозиторії. У кожній гілці слід виконати зміни у тих самих або пов’язаних частинах файлу. Це дозволить створити ситуацію, у якій виникає конфлікт під час злиття.

Далі необхідно зафіксувати зміни у кожній гілці у вигляді окремих комітів. Повідомлення комітів повинні відображати зміст виконаних змін. Це дозволяє краще орієнтуватися у процесі об’єднання.

Після цього необхідно виконати злиття однієї гілки в іншу. У разі відсутності конфліктів процес завершиться автоматично. Якщо ж конфлікти виникають, система повідомляє про це та позначає відповідні файли.

На наступному етапі необхідно відкрити файли, у яких виникли конфлікти, та проаналізувати їх вміст. Слід звернути увагу на спеціальні маркери, що відображають різні варіанти змін. Необхідно визначити, який варіант є правильним, або сформувати новий варіант, що поєднує необхідні зміни.

Після внесення змін необхідно видалити службові позначення конфлікту та зберегти файл. Далі потрібно додати файл до індексу та завершити процес злиття шляхом створення відповідного коміту.

Особливу увагу необхідно приділити перевірці результату. Потрібно переконатися, що код працює коректно, а зміни не призвели до порушення логіки програми. За можливості доцільно виконати тестування.

У процесі виконання роботи необхідно проаналізувати причини виникнення конфліктів. Це дозволяє зрозуміти, як уникати подібних ситуацій у майбутньому та покращити організацію роботи з репозиторієм.

Додатково слід звернути увагу на правила організації роботи з гілками. Регулярне оновлення гілок, використання невеликих змін і чітке розмежування задач дозволяють зменшити кількість конфліктів.

Після завершення роботи необхідно оцінити складність процесу злиття та ефективність використаних підходів. Це дозволяє сформувати практичні навички роботи з Git та підвищити якість організації процесу розробки.

Таким чином, виконання роботи дозволяє сформувати навички об’єднання змін, розв’язання конфліктів і забезпечення узгодженості програмного коду у процесі командної роботи.

Контрольні питання

1. Що таке злиття змін у Git?
2. Що таке конфлікт у репозиторії?
3. У яких випадках виникають конфлікти?
4. Як позначаються конфлікти у файлах?
5. Які етапи розв’язання конфлікту?
6. Чому важливо аналізувати контекст змін?
7. Які способи злиття існують у Git?
8. Що таке перебазування?
9. Як зменшити ймовірність виникнення конфліктів?
10. Чому необхідно перевіряти код після злиття?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис створених гілок;
– опис внесених змін;
– результати злиття;
– приклади конфліктів;
– опис процесу їх розв’язання;
– аналіз отриманих результатів;
– висновки.

Практична робота 12. Налаштування системи збирання та автоматизації проєкту

Мета роботи – ознайомитися з принципами автоматизації процесу розробки програмного забезпечення, навчитися налаштовувати систему збирання проєкту та організовувати автоматичне виконання основних операцій.

Ключові положення

Система збирання програмного забезпечення є сукупністю засобів, що автоматизують перетворення вихідного коду у виконуваний продукт. Вона охоплює компіляцію, об’єднання модулів, підключення бібліотек, тестування та підготовку до розгортання. Її використання зменшує кількість ручних дій і підвищує надійність розробки.

У сучасних проєктах система збирання забезпечує узгодженість середовища, визначаючи інструменти, параметри та послідовність операцій. Вона також керує залежностями між файлами, дозволяючи перезбирати лише необхідні частини проєкту.

Автоматизація передбачає виконання типових задач, зокрема компіляції, тестування та перевірки коду, що забезпечує повторюваність процесів. Сценарії збирання дозволяють виконувати всі операції однією командою, а засоби керування залежностями — коректно працювати зі сторонніми бібліотеками.

Інтеграція з системами керування версіями забезпечує автоматичний запуск збирання після змін у репозиторії, що є основою безперервної інтеграції. Під час збирання можуть запускатися тести, що дозволяє виявляти помилки на ранніх етапах.

Система збирання повинна бути зрозумілою, документованою та гнучкою, щоб адаптуватися до змін у проєкті. Таким чином, її налаштування забезпечує ефективність розробки та якість програмного продукту.

Практичне завдання

1. Ознайомитися з поняттям системи збирання програмного забезпечення.
2. Обрати інструмент для збирання проєкту.
3. Проаналізувати структуру проєкту.
4. Визначити залежності між файлами.
5. Створити сценарій збирання.
6. Налаштувати процес компіляції.
7. Налаштувати підключення бібліотек.
8. Реалізувати автоматичний запуск тестів.
9. Перевірити коректність роботи системи збирання.
10. Проаналізувати результати автоматизації.
11. Зробити висновки щодо ефективності налаштувань.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з принципами автоматизації процесу розробки та роллю систем збирання у програмних проєктах. Доцільно звернути увагу на те, що система збирання повинна не лише виконувати компіляцію, але й забезпечувати виконання супутніх задач, необхідних для підготовки програмного продукту.

На першому етапі необхідно обрати інструмент для збирання. Вибір залежить від мови програмування, структури проєкту та вимог до автоматизації. Після вибору інструмента слід ознайомитися з його основними можливостями та принципами роботи.

Далі необхідно проаналізувати структуру проєкту. Слід визначити розташування вихідних файлів, бібліотек, конфігураційних файлів та інших компонентів. Це дозволяє правильно налаштувати процес збирання.

На наступному етапі потрібно визначити залежності між файлами. Необхідно встановити, які файли впливають один на одного та у якій послідовності повинні виконуватися операції. Це є основою для побудови сценарію збирання.

Після цього необхідно створити сценарій збирання. У сценарії слід описати послідовність виконання операцій, включаючи компіляцію, об’єднання модулів та інші дії. Важливо забезпечити коректність виконання кожного етапу.

Далі необхідно налаштувати процес компіляції. Слід визначити параметри компілятора, шляхи до файлів та інші необхідні налаштування. У разі використання бібліотек потрібно забезпечити їх правильне підключення.

На наступному етапі необхідно реалізувати автоматичний запуск тестів. Це дозволяє перевіряти працездатність проєкту після кожного збирання. У разі виявлення помилок необхідно проаналізувати їх причини та внести відповідні зміни.

Після налаштування системи збирання необхідно перевірити її роботу. Слід виконати збирання проєкту та переконатися у правильності виконання всіх операцій. У разі виявлення помилок необхідно скоригувати налаштування.

Додатково слід звернути увагу на зручність використання системи. Доцільно забезпечити можливість запуску збирання однією командою, що значно спрощує роботу з проєктом.

Також необхідно проаналізувати ефективність автоматизації. Слід оцінити, наскільки налаштована система спрощує процес розробки, зменшує кількість ручних дій та підвищує якість програмного продукту.

Після завершення роботи необхідно зробити висновки щодо доцільності використання обраного інструмента та ефективності налаштування системи збирання.

Таким чином, виконання роботи дозволяє сформувати практичні навички налаштування систем автоматизації та організації процесу збирання програмного забезпечення.

Контрольні питання

1. Що таке система збирання?
2. Яке призначення автоматизації у розробці?
3. Що таке сценарій збирання?
4. Які задачі може виконувати система збирання?
5. Що таке залежності у проєкті?
6. Яка роль тестування у процесі збирання?
7. Що таке безперервна інтеграція?
8. Чому важлива автоматизація процесів?
9. Які переваги використання систем збирання?
10. Як оцінити ефективність автоматизації?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис обраного інструмента;
– структура проєкту;
– сценарій збирання;
– результати виконання;
– аналіз ефективності;
– висновки.

Практична робота 13. Організація взаємодії програмного коду з базою даних

Мета роботи – ознайомитися з принципами взаємодії програмного забезпечення з базами даних, навчитися організовувати доступ до даних, виконувати основні операції збереження, вибірки та обробки інформації у програмній системі.

Ключові положення

Взаємодія програмного коду з базою даних є ключовою складовою сучасних програмних систем, що забезпечує зберігання, обробку та керування даними. Від її організації залежать продуктивність, надійність і стабільність роботи системи.

База даних є структурованим сховищем, яке підтримує операції додавання, зміни, видалення та вибірки даних. Доступ до неї здійснюється через бібліотеки, драйвери або API, що дозволяє відокремити програмну логіку від конкретної реалізації сховища. Основним засобом роботи є запити, які повинні формуватися коректно для запобігання помилкам і втраті даних.

У програмних системах часто застосовується рівень абстракції, що ізолює роботу з даними та спрощує зміну бази даних. Важливими є правильна організація підключення, управління транзакціями та забезпечення цілісності даних.

Значну роль відіграє безпека, яка передбачає перевірку вхідних даних і обмеження доступу. Також необхідно враховувати продуктивність, оптимізуючи запити та структуру даних, і забезпечувати коректну обробку помилок.

У сучасних системах взаємодія з базою даних часто винесена в окремий рівень, що спрощує супроводження та повторне використання коду. Таким чином, правильна організація роботи з базою даних забезпечує ефективність і надійність програмного продукту.

Практичне завдання

1. Ознайомитися з принципами роботи баз даних.
2. Обрати базу даних для використання.
3. Створити структуру бази даних.
4. Реалізувати підключення до бази даних.
5. Реалізувати операції додавання даних.
6. Реалізувати операції вибірки даних.
7. Реалізувати операції оновлення даних.
8. Реалізувати операції видалення даних.
9. Організувати обробку помилок.
10. Перевірити коректність роботи.
11. Зробити висновки щодо ефективності реалізації.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з основами роботи баз даних та принципами організації доступу до даних у програмних системах. Доцільно

звернути увагу на роль бази даних у загальній архітектурі системи та способи взаємодії з нею.

На першому етапі необхідно обрати базу даних. Вибір може залежати від типу проєкту, обсягу даних та вимог до системи. Після цього слід створити структуру бази даних, що включає таблиці, поля та зв'язки між ними.

Далі необхідно реалізувати підключення до бази даних. Слід визначити параметри підключення та забезпечити коректне встановлення з'єднання. Важливо передбачити обробку можливих помилок під час підключення.

На наступному етапі потрібно реалізувати основні операції роботи з даними. Це включає додавання, вибірку, оновлення та видалення інформації. Кожна операція повинна бути реалізована з урахуванням вимог до системи.

Особливу увагу необхідно приділити перевірці вхідних даних. Слід забезпечити коректність інформації, що передається до бази даних, та уникати ситуацій, що можуть призвести до помилок.

Далі необхідно організувати обробку помилок. Програма повинна коректно реагувати на помилки, що виникають під час виконання операцій, і забезпечувати стабільну роботу системи.

Після реалізації основних функцій необхідно виконати тестування. Слід перевірити коректність виконання всіх операцій та відповідність отриманих результатів очікуваним.

Додатково слід звернути увагу на ефективність роботи з базою даних. Необхідно проаналізувати кількість виконуваних запитів та їх вплив на продуктивність системи.

Після завершення роботи необхідно зробити висновки щодо ефективності організації взаємодії з базою даних та можливих шляхів її покращення.

Таким чином, виконання роботи дозволяє сформувати практичні навички роботи з базами даних та організації доступу до інформації у програмних системах.

Контрольні питання

1. Що таке база даних?
2. Яке призначення бази даних у програмній системі?
3. Які операції виконуються над даними?
4. Що таке запит до бази даних?
5. Що таке транзакція?
6. Чому важлива обробка помилок?
7. Як забезпечується безпека даних?
8. Які фактори впливають на продуктивність?
9. Що таке рівень абстракції у роботі з базою даних?
10. Які переваги використання баз даних?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис обраної бази даних;
– структура бази даних;
– реалізація операцій;

– результати тестування;
– аналіз ефективності;
– висновки.

Практична робота 14. Реалізація обробки та валідації вхідних даних у програмному модулі

Мета роботи – ознайомитися з принципами обробки та перевірки вхідних даних, навчитися реалізовувати механізми валідації у програмному модулі та забезпечувати коректність і безпеку роботи програмної системи.

Ключові положення

Обробка вхідних даних є невід'ємною частиною функціонування будь-якої програмної системи. Дані можуть надходити від користувача, з інших програмних компонентів, із файлів або з мережевих джерел. Незалежно від джерела, ці дані можуть бути некоректними, неповними або навіть шкідливими, тому їх обробка повинна виконуватися з урахуванням перевірки достовірності та відповідності встановленим вимогам.

Валідація вхідних даних є процесом перевірки правильності, повноти та допустимості інформації, що надходить у систему. Вона спрямована на забезпечення того, щоб дані відповідали визначеним обмеженням, таким як формат, діапазон значень, обов'язковість заповнення та логічна узгодженість. Наявність механізмів валідації дозволяє запобігти виникненню помилок у програмі та забезпечує стабільність її роботи.

Одним із ключових аспектів є перевірка типів даних. Програма повинна переконатися, що значення мають очікуваний тип, наприклад числовий, текстовий або логічний. Невідповідність типів може призвести до помилок виконання або некоректних результатів обчислень. Тому перевірка типів є базовим рівнем валідації.

Важливим є також контроль діапазонів значень. Для числових даних це може бути перевірка на входження у допустимий інтервал, для текстових – обмеження довжини або перевірка на наявність допустимих символів. Такий контроль дозволяє уникнути помилок, пов'язаних із некоректними значеннями.

Окрему увагу слід приділити перевірці обов'язковості введення даних. Якщо певне поле є обов'язковим, програма повинна перевірити, що воно не є порожнім. Відсутність необхідних даних може призвести до порушення логіки роботи системи.

Валідація також може включати перевірку логічної узгодженості даних. Наприклад, якщо один параметр залежить від іншого, необхідно перевірити їх взаємозв'язок. Такий тип перевірки дозволяє виявляти складні помилки, що не можуть бути виявлені простими перевірками типів або діапазонів.

Обробка вхідних даних передбачає не лише їх перевірку, але й перетворення у формат, зручний для подальшого використання. Це може включати приведення типів, нормалізацію значень, видалення зайвих символів або інші операції. Такі перетворення дозволяють забезпечити узгодженість даних у системі.

Важливим аспектом є розмежування обробки та валідації. Валідація визначає, чи є дані допустимими, тоді як обробка виконує їх перетворення. Чітке розділення цих процесів дозволяє спростити структуру програмного коду та підвищити його зрозумілість.

Особливу увагу слід приділити безпеці. Некоректні або шкідливі дані можуть бути використані для атак на систему. Тому необхідно забезпечити перевірку даних перед їх використанням. Це включає обмеження допустимих значень, перевірку форматів і уникнення небезпечних операцій.

У процесі обробки вхідних даних важливо забезпечити коректну реакцію системи на помилки. У разі виявлення некоректних даних програма повинна повідомити користувача або інший компонент системи про помилку та не допускати виконання некоректних операцій.

Організація валідації може виконуватися на різних рівнях системи. Частина перевірок може виконуватися на рівні інтерфейсу користувача, інша – у серверній частині або у внутрішніх модулях. Такий підхід дозволяє забезпечити багаторівневий контроль даних.

Додатково слід зазначити, що повторне використання механізмів валідації дозволяє підвищити ефективність розробки. Загальні правила перевірки можуть бути винесені в окремі функції або модулі, що забезпечує їх використання у різних частинах системи.

Таким чином, реалізація обробки та валідації вхідних даних є важливим етапом розробки програмного забезпечення, що забезпечує коректність, надійність і безпеку роботи програмної системи.

Практичне завдання

1. Ознайомитися з принципами обробки вхідних даних.
2. Визначити джерела вхідних даних у програмному модулі.
3. Визначити вимоги до коректності даних.
4. Реалізувати перевірку типів даних.
5. Реалізувати перевірку діапазонів значень.
6. Реалізувати перевірку обов’язкових полів.
7. Реалізувати перевірку логічної узгодженості даних.
8. Реалізувати перетворення даних у необхідний формат.
9. Організувати обробку помилок.
10. Перевірити коректність роботи модуля.
11. Зробити висновки щодо ефективності реалізації.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з основами обробки та валідації даних. Доцільно звернути увагу на те, що перевірка даних є обов’язковим етапом їх використання у програмній системі.

На першому етапі необхідно визначити джерела вхідних даних. Це можуть бути дані, введені користувачем, отримані з файлів або передані іншими компонентами системи. Важливо врахувати всі можливі варіанти надходження інформації.

Далі необхідно визначити вимоги до коректності даних. Слід встановити, які значення є допустимими, які обмеження існують та які перевірки необхідно виконати.

На наступному етапі потрібно реалізувати перевірку типів даних. Програма повинна визначати, чи відповідають отримані значення очікуваному типу.

Після цього необхідно реалізувати перевірку діапазонів значень та обов’язковості полів. Це дозволяє забезпечити правильність даних та уникнути помилок.

Далі необхідно реалізувати перевірку логічної узгодженості. Слід враховувати взаємозв’язки між різними параметрами та перевіряти їх відповідність.

На наступному етапі потрібно виконати перетворення даних. Це може включати зміну формату, очищення даних або інші операції.

Після цього необхідно організувати обробку помилок. Програма повинна повідомляти про помилки та забезпечувати коректну роботу системи.

Додатково слід перевірити ефективність реалізованих механізмів. Необхідно переконатися, що всі перевірки виконуються коректно та не призводять до надмірного ускладнення коду.

Після завершення роботи необхідно зробити висновки щодо якості реалізації та можливих шляхів її вдосконалення.

Таким чином, виконання роботи дозволяє сформувати навички реалізації перевірки даних та забезпечення коректності роботи програмного забезпечення.

Контрольні питання

1. Що таке валідація даних?
2. Які види перевірок існують?
3. Чому важлива перевірка типів даних?
4. Що таке діапазон значень?
5. Чому важлива перевірка обов’язкових полів?
6. Що таке логічна узгодженість даних?
7. Яка роль обробки помилок?
8. Як забезпечується безпека даних?
9. Чому важливо розділяти обробку та валідацію?
10. Які переваги повторного використання механізмів валідації?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис вхідних даних;
– вимоги до даних;
– реалізація перевірок;
– результати тестування;
– аналіз ефективності;
– висновки.

ТЕМА 4. ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Практична робота 15. Розробка тестових сценаріїв для програмного модуля

Мета роботи – ознайомитися з принципами підготовки тестових сценаріїв для програмного модуля, навчитися визначати умови перевірки його роботи, формувати набори тестових даних та описувати очікувані результати виконання.

Ключові положення

Тестування програмного забезпечення є важливою складовою розробки, що дозволяє перевірити відповідність модуля вимогам, виявити дефекти та оцінити якість. Основним інструментом організації тестування є тестові сценарії, які визначають дії, вхідні дані, умови виконання та очікувані результати.

Тестовий сценарій є формалізованим описом перевірки функцій або поведінки модуля, що забезпечує повторюваність тестування. Його розробка базується на вимогах і повинна охоплювати як стандартні, так і помилкові ситуації, що дозволяє оцінити стійкість системи.

Сценарій зазвичай містить ідентифікатор, умови, вхідні дані, кроки виконання та очікуваний результат. Важливим є правильний вибір тестових даних, зокрема типових, граничних і некоректних значень. Для цього застосовуються підходи еквівалентного розбиття та аналізу граничних значень.

Необхідно розрізняти позитивні та негативні сценарії, а також враховувати початкові умови виконання. Очікувані результати повинні бути чіткими й однозначними. Сценарії використовуються як для ручного, так і для автоматизованого тестування.

Важливо забезпечити повноту покриття функцій модуля та пов’язувати сценарії з вимогами. Їх розробка сприяє глибшому розумінню системи та полегшує командну роботу і повторне тестування. Таким чином, тестові сценарії забезпечують системність перевірки та підвищують якість програмного продукту.

Практичне завдання

1. Ознайомитися з поняттям тестового сценарію.
2. Обрати програмний модуль для тестування.
3. Проаналізувати функції обраного модуля.
4. Визначити вимоги, які повинні бути перевірені.
5. Визначити позитивні сценарії тестування.
6. Визначити негативні сценарії тестування.
7. Підібрати тестові дані для кожного сценарію.
8. Сформулювати очікувані результати виконання.
9. Оформити набір тестових сценаріїв у структурованому вигляді.
10. Перевірити повноту охоплення основних функцій модуля.
11. Зробити висновки щодо якості підготовлених сценаріїв.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з основами тестування програмного забезпечення та роллю тестових сценаріїв у процесі перевірки якості програмного модуля. Доцільно звернути увагу на те, що тестовий сценарій повинен не просто фіксувати дію користувача або виклик функції, а відображати повну логіку перевірки з урахуванням умов виконання та очікуваних результатів.

На першому етапі необхідно обрати програмний модуль, для якого будуть розроблятися тестові сценарії. Доцільно використовувати модуль, що був реалізований у попередніх практичних роботах. Це дозволить забезпечити логічний зв'язок між роботами та виконати тестування на основі вже створеного програмного коду. Модуль повинен мати достатньо чіткі функції, щоб можна було виділити окремі перевірки.

Далі необхідно проаналізувати функціональність модуля. Слід визначити, які саме операції він виконує, які дані приймає на вхід, які результати повертає, які обмеження накладаються на його роботу. На цьому етапі доцільно також встановити, які помилкові ситуації можуть виникати під час використання модуля. Саме цей аналіз стане основою для подальшого формування тестових сценаріїв.

Після цього необхідно визначити перелік функцій або перевірок, які повинні бути охоплені тестуванням. Не слід обмежуватися лише основним позитивним шляхом виконання операцій. Доцільно одразу передбачити перевірку неправильних вхідних даних, відсутності обов'язкових параметрів, перевищення допустимих меж та інших умов, за яких модуль повинен або правильно відмовити у виконанні, або обробити помилку у визначений спосіб.

На наступному етапі потрібно сформувати позитивні тестові сценарії. Для кожного такого сценарію слід визначити, за яких умов модуль повинен працювати правильно, які дані подаються на вхід та який результат очікується на виході. Позитивні сценарії повинні охоплювати основні функції модуля та підтверджувати, що він виконує своє призначення.

Далі необхідно сформувати негативні тестові сценарії. У цих сценаріях потрібно описати ситуації, коли на вхід подаються некоректні, неповні або недопустимі дані. Для кожного випадку слід визначити, як саме модуль повинен реагувати: повідомленням про помилку, відмовою від виконання операції, поверненням спеціального значення або іншим передбаченим механізмом. Негативні сценарії є важливими для оцінювання стійкості модуля.

Після визначення типів сценаріїв необхідно підібрати тестові дані. Для цього доцільно враховувати не лише типові приклади, але й граничні значення. Якщо параметр має допустимий діапазон, слід перевірити значення на нижній межі, на верхній межі, трохи нижче і трохи вище цих меж. Якщо йдеться про текстові дані, варто перевірити порожні значення, мінімально допустиму довжину, максимально допустиму довжину та перевищення обмежень.

Особливу увагу необхідно приділити формулюванню очікуваних результатів. Для кожного сценарію потрібно чітко вказати, що саме вважатиметься правильним результатом. Наприклад, це може бути створення нового запису, зміна стану об'єкта, повернення певного повідомлення, виведення помилки або відсутність змін у даних. Очікуваний результат повинен бути конкретним, щоб його можна було однозначно порівняти з фактичним результатом під час виконання перевірки.

Далі необхідно оформити тестові сценарії у структурованому вигляді. Для кожного сценарію доцільно вказати ідентифікатор, назву, мету перевірки, початкові умови, тестові

дані, послідовність дій та очікуваний результат. Якщо цього вимагає формат роботи, можна додатково залишити поле для фактичного результату та висновку про успішність виконання тесту. Такий формат робить результати впорядкованими і зручними для використання.

Після підготовки сценаріїв доцільно виконати аналіз повноти тестування. Необхідно перевірити, чи охоплені всі основні функції модуля, чи враховані типові та нетипові випадки, чи достатньо перевірені обмеження щодо вхідних даних. Якщо певна функція не має відповідного сценарію, це свідчить про неповноту підготовленого набору перевірок.

У процесі виконання роботи важливо звертати увагу на реалістичність сценаріїв. Вони повинні відображати не абстрактні перевірки, а ситуації, які можуть виникати під час фактичного використання програмного модуля. Це робить тестування більш змістовним і практично орієнтованим.

Додатково доцільно проаналізувати, які зі створених сценаріїв у подальшому можуть бути автоматизовані. Якщо сценарій містить чітко визначені вхідні дані, однозначний очікуваний результат і повторювану послідовність дій, він є придатним для подальшої автоматизації. Такий підхід дозволяє розглядати розроблені сценарії як основу для наступних робіт, пов’язаних з автоматизованим тестуванням.

Після завершення підготовки сценаріїв необхідно сформулювати висновки. У висновках слід коротко охарактеризувати обраний модуль, оцінити повноту розроблених тестових сценаріїв та зазначити, наскільки підготовлений набір перевірок дозволяє оцінити якість роботи програмного модуля.

Таким чином, послідовне виконання роботи дозволяє сформувати практичні навички аналізу функцій програмного модуля, підготовки тестових даних, розробки позитивних і негативних тестових сценаріїв та структурованого документування результатів тестування.

Контрольні питання

1. Що таке тестовий сценарій?
2. Яке призначення тестових сценаріїв у процесі тестування?
3. Які елементи повинен містити тестовий сценарій?
4. Чим відрізняються позитивні та негативні сценарії?
5. Чому важливо визначати початкові умови виконання сценарію?
6. Яку роль відіграють тестові дані?
7. Що таке граничні значення у тестуванні?
8. Чому очікуваний результат повинен бути однозначним?
9. Як перевірити повноту набору тестових сценаріїв?
10. Чому тестові сценарії доцільно пов’язувати з вимогами до модуля?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис обраного програмного модуля;
– перелік функцій, що підлягають тестуванню;
– набір тестових сценаріїв;
– опис тестових даних;
– очікувані результати виконання;

– аналіз повноти тестування;
– висновки.

Практична робота 16. Реалізація модульного тестування

Мета роботи – ознайомитися з принципами модульного тестування програмного забезпечення, навчитися реалізовувати автоматизовані тести для окремих функцій і компонентів програмного модуля та оцінювати коректність їх роботи.

Ключові положення

Модульне тестування є одним із базових рівнів перевірки програмного забезпечення, що спрямований на тестування окремих функцій, процедур або класів незалежно від інших частин системи. Його основною метою є виявлення помилок на ранніх етапах розробки, коли виправлення дефектів є найменш витратним. Модульне тестування дозволяє перевірити коректність реалізації логіки, обробку вхідних даних та відповідність результатів визначеним вимогам.

Під модулем у контексті тестування розуміють найменшу логічно завершену частину програми, яка може бути протестована ізольовано. Це може бути окрема функція, метод класу або невеликий компонент. Ізоляція модуля є важливою умовою модульного тестування, оскільки дозволяє зосередитися на перевірці конкретної логіки без впливу інших частин системи.

Модульні тести зазвичай реалізуються у вигляді програмного коду, що автоматично виконує перевірку роботи функцій. Такий підхід дозволяє багаторазово запускати тести, що є важливим під час внесення змін у програму. Автоматизовані тести забезпечують стабільність системи та дозволяють швидко виявляти помилки після модифікації коду.

Кожен модульний тест повинен перевіряти окремий аспект роботи функції. Для цього визначаються вхідні дані, виконується виклик функції та перевіряється відповідність отриманого результату очікуваному значенню. Якщо результат не відповідає очікуванню, тест вважається невдалим, що сигналізує про наявність помилки у коді.

Важливою характеристикою модульного тестування є незалежність тестів. Кожен тест повинен виконуватися незалежно від інших і не залежати від їх результатів. Це дозволяє легко локалізувати помилки та спрощує аналіз результатів тестування.

Під час модульного тестування широко застосовуються заглушки та імітатори, що дозволяють замінити зовнішні залежності. Якщо функція залежить від інших компонентів, їх можна замінити спрощеними реалізаціями, що повертають контрольовані результати. Це забезпечує ізоляцію тестованого модуля.

Важливим аспектом є покриття коду тестами. Покриття визначає, яка частина програмного коду перевіряється під час тестування. Чим вищий рівень покриття, тим більше гарантій того, що код працює коректно. Однак доцільно прагнути не лише до кількісного покриття, але й до змістовної перевірки різних сценаріїв роботи.

Модульні тести повинні охоплювати як позитивні, так і негативні сценарії. Позитивні сценарії перевіряють правильну роботу функцій за коректних умов, тоді як негативні – реакцію на помилки, некоректні дані або виняткові ситуації. Такий підхід дозволяє оцінити стійкість програмного модуля.

Організація модульного тестування передбачає використання спеціальних інструментів або бібліотек. Вони дозволяють структурувати тести, автоматично запускати їх та отримувати результати у зручному вигляді. Це значно спрощує процес перевірки та аналізу результатів.

Важливою практикою є регулярний запуск тестів під час розробки. Після внесення змін у код доцільно одразу перевіряти його працездатність за допомогою модульних тестів. Це дозволяє своєчасно виявляти помилки та запобігати їх накопиченню.

Модульне тестування також сприяє підвищенню якості програмного коду. Наявність тестів стимулює розробника до створення більш зрозумілих і структурованих функцій, що легко піддаються перевірці. Це позитивно впливає на загальну якість програмного продукту.

Додатково слід зазначити, що модульні тести можуть використовуватися як частина документації. Вони демонструють, як саме повинні працювати функції, і можуть слугувати прикладом їх використання.

Таким чином, реалізація модульного тестування є важливим етапом забезпечення якості програмного забезпечення, що дозволяє виявляти помилки на ранніх етапах, забезпечувати стабільність системи та підвищувати ефективність процесу розробки.

Практичне завдання

1. Ознайомитися з поняттям модульного тестування.
2. Обрати програмний модуль для тестування.
3. Визначити функції, що підлягають перевірці.
4. Розробити тестові випадки для кожної функції.
5. Реалізувати модульні тести.
6. Виконати перевірку позитивних сценаріїв.
7. Виконати перевірку негативних сценаріїв.
8. Виконати запуск тестів.
9. Проаналізувати результати тестування.
10. Виправити виявлені помилки (за наявності).
11. Зробити висновки щодо якості модуля.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з принципами модульного тестування та його місцем у процесі розробки програмного забезпечення. Доцільно звернути увагу на те, що модульне тестування виконується на ранніх етапах і дозволяє виявити помилки ще до інтеграції компонентів.

На першому етапі необхідно обрати програмний модуль для тестування. Доцільно використовувати модуль, реалізований у попередніх роботах. Це дозволить перевірити реальний код і оцінити його якість.

Далі необхідно визначити функції, що підлягають тестуванню. Слід виділити ключові операції модуля та визначити, які аспекти їх роботи необхідно перевірити.

На наступному етапі потрібно розробити тестові випадки. Для кожної функції слід визначити вхідні дані, очікуваний результат та умови виконання. Важливо охопити як типові, так і граничні випадки.

Після цього необхідно реалізувати модульні тести у вигляді програмного коду. Слід використовувати інструменти або бібліотеки, що дозволяють автоматизувати процес тестування.

Особливу увагу необхідно приділити ізоляції тестів. Якщо функція залежить від інших компонентів, доцільно використати заглушки або імітатори.

Далі необхідно виконати запуск тестів і проаналізувати результати. У разі виявлення помилок слід визначити їх причини та внести необхідні зміни у код.

Після виправлення помилок необхідно повторно виконати тестування, щоб переконатися у правильності внесених змін.

Додатково слід оцінити повноту покриття тестами. Необхідно перевірити, чи всі основні функції модуля мають відповідні тести.

Після завершення роботи необхідно зробити висновки щодо якості реалізованого модуля та ефективності модульного тестування.

Таким чином, виконання роботи дозволяє сформувати практичні навички автоматизованого тестування та забезпечення якості програмного забезпечення.

Контрольні питання

1. Що таке модульне тестування?
2. Яке його призначення?
3. Що таке модуль у контексті тестування?
4. Які властивості повинні мати модульні тести?
5. Що таке покриття коду?
6. Чому важлива ізоляція тестів?
7. Що таке заглушки та імітатори?
8. Які сценарії повинні перевірятися?
9. Яка роль автоматизації у тестуванні?
10. Як оцінити якість тестів?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис програмного модуля;
– перелік тестових випадків;
– реалізація тестів;
– результати виконання;
– аналіз помилок;
– висновки.

Практична робота 17. Проведення інтеграційного тестування та аналіз результатів

Мета роботи – ознайомитися з принципами інтеграційного тестування програмного забезпечення, навчитися перевіряти взаємодію між модулями системи та виконувати аналіз отриманих результатів.

Ключові положення

Інтеграційне тестування є етапом перевірки програмного забезпечення, на якому оцінюється коректність взаємодії між окремими модулями системи. На відміну від модульного тестування, яке зосереджене на перевірці окремих функцій або компонентів, інтеграційне тестування дозволяє виявити помилки, що виникають під час об'єднання частин системи у єдину структуру.

Основною метою інтеграційного тестування є перевірка коректності обміну даними між модулями, узгодженості інтерфейсів та правильності виконання спільних операцій. У процесі взаємодії модулі можуть обмінюватися даними, викликати функції один одного або використовувати спільні ресурси. Помилки на цьому рівні часто пов'язані з невідповідністю форматів даних, неправильним порядком виклику або некоректною обробкою результатів.

Інтеграційне тестування може виконуватися за різними підходами. Поступова інтеграція передбачає поетапне об'єднання модулів із перевіркою на кожному кроці. Такий підхід дозволяє локалізувати помилки та спрощує їх аналіз. Альтернативний підхід полягає у одночасній інтеграції всіх модулів, що може бути швидшим, але ускладнює виявлення причин помилок.

Важливим аспектом є визначення сценаріїв інтеграційного тестування. Такі сценарії повинні охоплювати типові варіанти взаємодії між модулями, а також ситуації, що можуть призвести до помилок. Наприклад, слід перевіряти правильність передавання даних, обробку некоректних значень, реакцію на відсутність необхідних ресурсів та інші критичні умови.

Інтеграційне тестування тісно пов'язане з поняттям інтерфейсів. Саме інтерфейси визначають спосіб взаємодії між модулями, тому їх коректність є ключовою умовою успішної інтеграції. Якщо інтерфейси реалізовані некоректно або не відповідають специфікації, це призводить до помилок навіть у разі правильної роботи окремих модулів.

Під час інтеграційного тестування часто використовуються допоміжні засоби, такі як заглушки та імітатори. Вони дозволяють замінити відсутні або ще не реалізовані компоненти системи, що дає змогу виконувати тестування незалежно від стану розробки інших частин.

Важливим аспектом є організація тестових даних. Дані повинні бути підібрані таким чином, щоб перевірити різні варіанти взаємодії між модулями. Це включає як стандартні сценарії, так і граничні випадки та помилкові ситуації.

Результати інтеграційного тестування повинні бути зафіксовані та проаналізовані. Аналіз результатів дозволяє визначити причини виникнення помилок, оцінити їх вплив на систему та визначити шляхи їх усунення. Важливо не лише виявити помилки, але й зрозуміти їх природу.

Окрему увагу слід приділити повторному тестуванню після внесення змін. Після виправлення помилок необхідно перевірити, чи не виникли нові проблеми та чи збережена коректність роботи системи.

Інтеграційне тестування є важливим етапом підготовки програмного забезпечення до системного тестування та введення в експлуатацію. Воно дозволяє переконатися, що система функціонує як єдине ціле та відповідає визначеним вимогам.

Додатково слід зазначити, що інтеграційне тестування сприяє покращенню якості архітектури системи. Виявлені проблеми можуть свідчити про недоліки у проєктуванні, що потребують коригування.

Таким чином, проведення інтеграційного тестування та аналіз його результатів є важливими етапами забезпечення якості програмного забезпечення, що дозволяють перевірити взаємодію компонентів і підвищити надійність системи.

Практичне завдання

1. Ознайомитися з поняттям інтеграційного тестування.
2. Обрати модулі для перевірки взаємодії.
3. Проаналізувати інтерфейси між модулями.
4. Розробити сценарії інтеграційного тестування.
5. Підготувати тестові дані.
6. Виконати інтеграцію модулів.
7. Провести тестування взаємодії.
8. Зафіксувати результати виконання.
9. Виявити помилки.
10. Проаналізувати причини їх виникнення.
11. Зробити висновки щодо якості інтеграції.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з принципами інтеграційного тестування та його відмінностями від модульного тестування. Доцільно звернути увагу на те, що основною метою є перевірка взаємодії між компонентами, а не окремих функцій.

На першому етапі необхідно обрати декілька модулів, що взаємодіють між собою. Доцільно використовувати модулі, реалізовані у попередніх роботах. Це дозволить виконати тестування на основі реальної програмної системи.

Далі необхідно проаналізувати інтерфейси взаємодії між модулями. Слід визначити, які дані передаються, які функції викликаються та які результати очікуються.

На наступному етапі потрібно розробити сценарії інтеграційного тестування. Сценарії повинні охоплювати основні варіанти взаємодії, а також можливі помилкові ситуації.

Після цього необхідно підготувати тестові дані. Дані повинні бути підібрані таким чином, щоб перевірити різні аспекти взаємодії.

Далі необхідно виконати інтеграцію модулів і провести тестування. У процесі тестування слід фіксувати результати виконання кожного сценарію.

У разі виявлення помилок необхідно проаналізувати їх причини. Це може бути пов’язано з неправильними інтерфейсами, некоректною передачею даних або помилками у логіці роботи.

Після цього необхідно зробити висновки щодо якості інтеграції та визначити можливі шляхи покращення.

Таким чином, виконання роботи дозволяє сформувати навички інтеграційного тестування та аналізу результатів взаємодії програмних компонентів.

Контрольні питання

1. Що таке інтеграційне тестування?
2. Чим воно відрізняється від модульного тестування?
3. Які задачі вирішує інтеграційне тестування?
4. Які підходи до інтеграції існують?
5. Що таке інтерфейс взаємодії?
6. Які типи помилок можуть виникати?
7. Яка роль тестових даних?
8. Що таке заглушки та імітатори?
9. Чому важливий аналіз результатів?
10. Яка роль повторного тестування?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис модулів;
– сценарії тестування;
– результати виконання;
– аналіз помилок;
– висновки.

Практична робота 18. Виконання статичного аналізу коду та оформлення звіту про дефекти

Мета роботи – ознайомитися з принципами статичного аналізу програмного коду, навчитися виявляти потенційні дефекти без виконання програми та оформлювати результати перевірки у вигляді звіту про дефекти.

Ключові положення

Забезпечення якості програмного забезпечення передбачає використання різних підходів до виявлення помилок і недоліків програмного коду. Одним із важливих засобів такої перевірки є статичний аналіз, який виконується без запуску програми. На відміну від динамічного аналізу, що базується на спостереженні за поведінкою програмного продукту під час виконання, статичний аналіз орієнтований на дослідження структури вихідного коду, його синтаксису, логічної узгодженості, відповідності правилам оформлення та потенційних ризиків, які можуть призвести до дефектів у майбутньому.

Статичний аналіз дозволяє виявляти широкий спектр проблем ще до етапу запуску програми. До таких проблем можуть належати синтаксичні помилки, невикористані змінні, недосяжні ділянки коду, потенційні переповнення, порушення правил типізації, неправильна обробка виняткових ситуацій, дублювання фрагментів коду, надмірна складність окремих функцій та інші недоліки, що ускладнюють супроводження і знижують надійність програмної системи. Саме тому статичний аналіз розглядається як важлива складова процесу технічного контролю якості.

Однією з переваг статичного аналізу є можливість раннього виявлення помилок. Якщо дефект виявляється на ранньому етапі, його усунення зазвичай потребує менше часу і ресурсів, ніж у випадку, коли проблема проявляється вже після інтеграції або під час експлуатації системи. Крім того, рання перевірка дозволяє запобігти накопиченню дефектів, які у складних проєктах можуть поширюватися на інші модулі та викликати вторинні помилки.

Статичний аналіз може виконуватися вручну або із застосуванням спеціалізованих інструментальних засобів. Ручний аналіз передбачає перегляд програмного коду розробником, тестувальником або групою фахівців із метою виявлення підозрілих конструкцій, логічних помилок і відхилень від прийнятих стандартів. Інструментальний аналіз виконується за допомогою програмних засобів, які автоматично перевіряють код відповідно до визначених правил. У практиці розробки обидва підходи часто поєднуються, оскільки автоматичні засоби забезпечують швидкість перевірки, а ручний аналіз дозволяє глибше оцінити зміст програмної логіки.

Важливим аспектом статичного аналізу є перевірка стилю і структури коду. Хоча порушення стилістичних рекомендацій не завжди призводять до безпосередніх збоїв у роботі програми, вони можуть суттєво ускладнювати читання, розуміння і супроводження програмного коду. Надмірно довгі функції, неінформативні назви змінних, відсутність логічного структурування, зайві вкладення умов і циклів, невиправдане дублювання фрагментів коду знижують якість програмного продукту та підвищують ймовірність подальших помилок.

Окрему роль статичний аналіз відіграє у виявленні потенційно небезпечних конструкцій. Це можуть бути ділянки коду, у яких використовуються неініціалізовані змінні, можливе звернення за межі масиву, присутній ризик ділення на нуль, не перевіряються результати виклику функцій або відсутня коректна обробка виняткових ситуацій. У багатьох випадках такі проблеми не проявляються одразу, але створюють передумови для появи дефектів під час виконання програми або за певних комбінацій вхідних даних.

Статичний аналіз пов’язаний також із перевіркою відповідності коду прийнятим стандартам і правилам розробки. У межах навчального або виробничого проєкту можуть застосовуватися певні вимоги до структури файлів, іменування сутностей, коментування, організації модулів та оформлення помилок. Дотримання єдиних правил сприяє підвищенню узгодженості командної роботи та полегшує перегляд коду різними учасниками проєкту. Тому результати статичного аналізу нерідко відображають не лише технічні дефекти, а й порушення організаційних вимог до розробки.

Суттєве значення має оцінювання складності коду. Якщо окремі функції містять велику кількість умовних переходів, вкладених конструкцій і взаємопов’язаних операцій, це свідчить про підвищену складність і може бути ознакою проблемної реалізації. Такий код важче перевіряти, тестувати і модифікувати. Статичний аналіз дозволяє виявляти подібні фрагменти та рекомендувати їх спрощення або декомпозицію на менші логічно завершені частини.

Результати статичного аналізу доцільно оформлювати у вигляді звіту про дефекти. Такий звіт є структурованим документом, у якому фіксуються виявлені недоліки програмного коду, місце їх знаходження, короткий опис проблеми, рівень критичності, можливі наслідки та рекомендації щодо усунення. Наявність звіту дозволяє впорядкувати результати перевірки, спростити подальшу роботу з виправлення помилок та забезпечити контроль за усуненням дефектів.

Під дефектом у межах такої роботи доцільно розуміти будь-яке виявлене відхилення від очікуваної якості коду, яке може вплинути на коректність, надійність, безпеку, продуктивність або зручність супроводження програми. Це означає, що до звіту можуть потрапляти як явні помилки, так і потенційно небезпечні або неякісні рішення. Водночас важливо розрізняти суттєві дефекти та незначні зауваження, оскільки це впливає на пріоритетність їх усунення.

Рівень критичності дефекту визначається його можливим впливом на роботу системи. Критичними можна вважати такі дефекти, що можуть призвести до аварійного завершення програми, втрати даних, порушення безпеки або неможливості виконання основної функції. Середні та незначні дефекти можуть стосуватися логічних неточностей, ризиків некоректної роботи за окремих умов, порушень стилю або недоліків структури коду. Правильне визначення критичності дозволяє раціонально планувати виправлення.

Під час оформлення звіту про дефекти важливо забезпечити однозначність формулювань. Для кожного дефекту бажано вказати ідентифікатор, файл або модуль, рядок або фрагмент коду, короткий опис проблеми, пояснення її сутності, категорію, рівень критичності та пропонований спосіб усунення. Чим точніше і зрозуміліше оформлений звіт, тим легше розробнику виправити виявлений недолік без додаткових уточнень.

Важливим етапом є також аналіз причин виникнення дефектів. Якщо виявлені помилки мають системний характер, наприклад повторюються в різних модулях або пов’язані з однаковим типом недоліків, це може свідчити про проблеми у стилі програмування, недостатнє опрацювання вимог, низький рівень контролю коду або відсутність усталених правил розробки. У такому випадку статичний аналіз дає змогу не лише виявити окремі недоліки, а й оцінити загальний стан якості програмного продукту.

Статичний аналіз не замінює інші види перевірки, але суттєво доповнює їх. Він не дозволяє безпосередньо визначити, як програма поводиться під час виконання, проте забезпечує можливість виявлення великої кількості потенційних проблем ще до запуску системи. У поєднанні з модульним, інтеграційним і системним тестуванням статичний аналіз формує більш повне уявлення про якість програмного забезпечення.

Додатково слід зазначити, що регулярне застосування статичного аналізу сприяє підвищенню дисципліни програмування. Якщо розробник знає, що код буде перевірятися не лише на працездатність, а й на структурну якість, це стимулює більш відповідальне ставлення до оформлення, організації та внутрішньої логіки програмного модуля. У довгостроковій перспективі це позитивно впливає на весь процес створення програмного продукту.

Таким чином, статичний аналіз коду та оформлення звіту про дефекти є важливими елементами забезпечення якості програмного забезпечення, що дозволяють своєчасно виявляти проблемні фрагменти, оцінювати їх значущість і формувати обґрунтовані рекомендації щодо вдосконалення програмного коду.

Практичне завдання

1. Ознайомитися з поняттям статичного аналізу програмного коду.
2. Обрати програмний модуль або фрагмент коду для перевірки.
3. Визначити критерії та правила аналізу.
4. Виконати перевірку коду вручну або із застосуванням інструментального засобу.
5. Виявити синтаксичні, логічні та структурні недоліки.

6. Виявити потенційно небезпечні конструкції у коді.
7. Проаналізувати відповідність коду вимогам до стилю та оформлення.
8. Систематизувати виявлені дефекти.
9. Визначити рівень критичності кожного дефекту.
10. Оформити звіт про дефекти.
11. Сформулювати рекомендації щодо усунення виявлених недоліків.
12. Зробити висновки щодо якості проаналізованого коду.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з призначенням статичного аналізу та його місцем у процесі забезпечення якості програмного забезпечення. Доцільно звернути увагу на те, що така перевірка виконується без запуску програми, а отже основна увага приділяється аналізу структури, логіки та оформлення коду. Важливо розуміти, що метою роботи є не лише формальне виявлення окремих зауважень, а змістовна оцінка якості програмного модуля.

На першому етапі необхідно обрати програмний модуль, файл або фрагмент коду, який буде піддано аналізу. Доцільно використовувати код, створений у попередніх практичних роботах, оскільки це дозволить зберегти логічний зв’язок між завданнями дисципліни. Обраний фрагмент повинен бути достатньо змістовним, щоб у ньому можна було оцінити не лише синтаксичну правильність, а й структуру, стиль та якість програмних рішень.

Далі необхідно визначити критерії аналізу. Перед початком перевірки слід встановити, на які саме аспекти буде звернено увагу. Це можуть бути коректність оголошення змінних, ініціалізація даних, організація умовних конструкцій, повторення однакових фрагментів коду, дотримання правил іменування, наявність або відсутність перевірки помилкових ситуацій, логічна завершеність функцій, зручність читання та інші характеристики. Наявність попередньо визначених критеріїв робить аналіз послідовним і обґрунтованим.

На наступному етапі необхідно виконати безпосередню перевірку коду. Якщо використовується ручний підхід, слід уважно переглянути програмний код по блоках, оцінюючи зміст кожної функції, правильність використання змінних, доцільність окремих конструкцій і потенційні ризики. Якщо застосовується програмний засіб статичного аналізу, необхідно зафіксувати обраний інструмент, виконати перевірку та проаналізувати отримані попередження. При цьому не слід безумовно сприймати всі повідомлення інструмента як остаточні дефекти. Кожне з них необхідно оцінити змістовно.

Під час аналізу доцільно окремо звернути увагу на синтаксичні та структурні недоліки. Потрібно перевірити, чи не містить код зайвих, невикористаних або помилково оголошених змінних, чи не є окремі ділянки недосяжними, чи не використовуються сумнівні конструкції, що можуть ускладнити подальше супроводження. Якщо у коді присутні надто великі функції або складні вкладені конструкції, це також варто зафіксувати як ознаку зниженої якості структури.

Особливу увагу слід приділити логічним ризикам. Необхідно перевірити, чи обробляються всі варіанти умов, чи не пропущено перевірку виняткових ситуацій, чи враховано можливість некоректних вхідних даних, чи не виникає ризик використання неініціалізованих значень. Навіть якщо такі проблеми не проявляються явно у вигляді

помилки компіляції, вони можуть бути джерелом майбутніх дефектів, тому повинні фіксуватися у звіті.

Далі необхідно оцінити відповідність коду правилам стилю та оформлення. Слід перевірити логічність імен змінних, функцій і класів, однорідність структури програмного коду, наявність надмірно довгих фрагментів, дублювання, нерівномірного форматування або неочевидних ділянок, що ускладнюють читання. Якщо у межах курсу використовуються певні правила оформлення коду, їх дотримання також повинно бути враховане під час аналізу.

Після виявлення недоліків необхідно виконати їх систематизацію. Для цього доцільно згрупувати дефекти за типами, наприклад синтаксичні, логічні, структурні, стилістичні, потенційно небезпечні. Така систематизація спрощує подальше оформлення результатів і дозволяє краще оцінити характер проблем у проаналізованому коді. Якщо виявлені дефекти повторюються, це також варто відзначити як ознаку системного характеру недоліків.

На наступному етапі потрібно визначити рівень критичності кожного дефекту. Для цього слід оцінити, наскільки виявлена проблема може вплинути на працездатність, надійність, безпеку або супроводжуваність програми. Необхідно розмежувати суттєві дефекти, що можуть призвести до неправильного функціонування, та зауваження меншого рівня, які не спричиняють негайних збоїв, але негативно впливають на якість коду.

Після цього необхідно оформити звіт про дефекти. У звіті бажано для кожного дефекту навести його ідентифікатор, місце розташування, короткий заголовок, опис сутності проблеми, рівень критичності та рекомендацію щодо усунення. Якщо це доречно, можна також наводити короткий фрагмент коду або посилання на відповідну функцію. Оформлення звіту повинно бути чітким і зручним для подальшого використання.

У процесі підготовки звіту важливо формулювати описи дефектів точно і без двозначностей. Наприклад, замість загального формулювання у коді є помилка слід вказувати конкретну проблему, наприклад не перевіряється результат введення, змінна може бути використана без попередньої ініціалізації, функція містить дубльовану логіку або відсутня обробка помилкового значення параметра. Такий підхід робить звіт більш практичним.

Додатково доцільно у підсумковій частині роботи виконати короткий аналіз загального стану коду. Необхідно оцінити, які типи дефектів переважають, чи є код загалом структурованим, чи присутні системні проблеми, пов’язані з організацією програмного модуля. Це дозволяє перейти від опису окремих недоліків до більш узагальненого висновку щодо якості програмного рішення.

Після завершення перевірки необхідно сформулювати висновки. У висновках слід коротко охарактеризувати проаналізований код, кількість і типи виявлених дефектів, оцінити їх вплив на якість програми та зазначити, які зміни є першочерговими для покращення програмного модуля.

Таким чином, послідовне виконання роботи дозволяє сформувати практичні навички оцінювання якості програмного коду без його запуску, виявлення дефектів різного характеру та оформлення результатів аналізу у вигляді структурованого звіту.

Контрольні питання

1. Що таке статичний аналіз програмного коду?
2. Чим статичний аналіз відрізняється від динамічного?

3. Які типи недоліків дозволяє виявити статичний аналіз?
4. Які переваги має виявлення дефектів на ранніх етапах розробки?
5. У чому полягає різниця між ручним та інструментальним статичним аналізом?
6. Що таке дефект у програмному коді?
7. Які критерії можуть використовуватися під час статичного аналізу?
8. Чому важливо визначати рівень критичності дефекту?
9. Які відомості доцільно включати до звіту про дефекти?
10. Чому результати статичного аналізу необхідно систематизувати?

Оформлення документації та підготовка релізу програмного продукту
– назва практичної роботи;
– мета виконання;
– опис проаналізованого програмного модуля;
– критерії статичного аналізу;
– перелік виявлених дефектів;
– оцінка критичності дефектів;
– звіт про дефекти;
– рекомендації щодо усунення недоліків;
– висновки.

Тема 5. Управління проєктами та супроводження програмних продуктів

Практична робота 19. Побудова плану реалізації програмного проєкту з оцінюванням трудомісткості

Мета роботи – ознайомитися з принципами планування розробки програмного забезпечення, навчитися формувати план реалізації проєкту, визначати етапи виконання робіт та оцінювати трудомісткість задач.

Ключові положення

Планування реалізації програмного проєкту є важливим етапом організації процесу розробки, що визначає послідовність виконання робіт, розподіл задач, терміни виконання та необхідні ресурси. Від якості планування залежить ефективність розробки, своєчасність виконання завдань та здатність команди досягти поставлених цілей.

План реалізації проєкту відображає структуру робіт, які необхідно виконати для створення програмного продукту. До таких робіт належать аналіз вимог, проєктування, реалізація, тестування, інтеграція, документування та підготовка до впровадження. Кожна з цих частин може деталізуватися до рівня окремих задач, що забезпечує більш точне планування.

Одним із основних інструментів планування є декомпозиція робіт. Складний проєкт поділяється на менші задачі, які легше оцінити та виконати. Такий підхід дозволяє підвищити керованість проєкту, спростити контроль виконання та забезпечити прозорість процесу розробки.

Важливим аспектом є визначення послідовності виконання задач. Деякі роботи можуть виконуватися паралельно, тоді як інші залежать від результатів попередніх етапів. Встановлення залежностей між задачами дозволяє сформувати логічно узгоджений план реалізації.

Оцінювання трудомісткості передбачає визначення обсягу зусиль, необхідних для виконання кожної задачі. Трудомісткість може вимірюватися у годинах або умовних одиницях і відображає складність виконання роботи. Точність оцінювання залежить від досвіду розробника, чіткості вимог та рівня деталізації задач.

Існують різні підходи до оцінювання трудомісткості. Один із них базується на експертній оцінці, коли розробник визначає приблизний час виконання задачі. Інший підхід передбачає використання аналогій із попередніми проєктами. Також застосовуються методи деталізації задач, коли складна задача розбивається на менші частини, кожна з яких оцінюється окремо.

Під час планування важливо враховувати ризики. Невизначеність вимог, технічні складнощі, обмеження ресурсів або зовнішні фактори можуть впливати на терміни виконання робіт. Тому доцільно передбачати резерв часу для компенсації можливих відхилень від плану.

План реалізації повинен бути реалістичним і узгодженим із можливостями команди. Надто оптимістичні оцінки можуть призвести до зриву термінів, тоді як надто завищені оцінки знижують ефективність використання ресурсів. Тому важливо досягти балансу між точністю оцінки та реалістичністю плану.

Важливим елементом є контроль виконання плану. У процесі розробки необхідно відстежувати виконання задач, порівнювати фактичні результати з плановими та за потреби коригувати план. Це дозволяє своєчасно реагувати на відхилення та забезпечувати стабільний хід проєкту.

Планування також передбачає визначення відповідальності за виконання задач. У командній розробці важливо розподілити задачі між учасниками, враховуючи їх компетенції та завантаженість. Це дозволяє підвищити ефективність роботи та уникнути перевантаження окремих розробників.

Окрему увагу слід приділити документуванню плану. План реалізації повинен бути оформлений у зрозумілому вигляді, що дозволяє використовувати його як основу для організації роботи. Документований план полегшує взаємодію між учасниками проєкту та забезпечує прозорість процесу.

Додатково слід зазначити, що планування є динамічним процесом. У міру розвитку проєкту можуть змінюватися вимоги, з’являтися нові задачі або уточнюватися оцінки. Тому план повинен допускати коригування без порушення загальної структури.

Таким чином, побудова плану реалізації програмного проєкту та оцінювання трудомісткості є важливими складовими управління розробкою, що забезпечують ефективну організацію роботи, контроль виконання задач та досягнення поставлених цілей.

Практичне завдання

1. Ознайомитися з принципами планування програмних проєктів.
2. Обрати програмний проєкт для планування.
3. Визначити основні етапи реалізації.
4. Виконати декомпозицію проєкту на задачі.
5. Визначити залежності між задачами.
6. Оцінити трудомісткість кожної задачі.
7. Визначити загальну трудомісткість проєкту.
8. Сформувати план реалізації.
9. Проаналізувати можливі ризики.
10. Оцінити реалістичність побудованого плану.
11. Зробити висновки щодо ефективності планування.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з основами планування програмних проєктів та роллю оцінювання трудомісткості у процесі розробки. Доцільно звернути увагу на те, що планування повинно базуватися на вимогах до системи та враховувати всі етапи її створення.

На першому етапі необхідно обрати програмний проєкт. Доцільно використовувати проєкт, що розроблявся у попередніх роботах. Це дозволить сформувати план на основі вже відомої структури системи.

Далі необхідно визначити основні етапи реалізації. Слід виділити такі етапи, як аналіз вимог, проєктування, реалізація, тестування та інтеграція.

На наступному етапі потрібно виконати декомпозицію проєкту на задачі. Кожен етап слід деталізувати до рівня окремих робіт.

Після цього необхідно визначити залежності між задачами. Слід встановити, які задачі можуть виконуватися паралельно, а які потребують завершення попередніх.

Далі необхідно оцінити трудомісткість кожної задачі. Для цього слід визначити приблизний час виконання кожної роботи.

Після оцінювання необхідно визначити загальну трудомісткість проєкту та сформувати план реалізації.

На наступному етапі слід проаналізувати можливі ризики та врахувати їх у плані.

Додатково необхідно оцінити реалістичність побудованого плану та за потреби внести коригування.

Після завершення роботи необхідно зробити висновки щодо ефективності планування.

Таким чином, виконання роботи дозволяє сформувати навички планування та оцінювання трудомісткості програмних проєктів.

Контрольні питання

1. Що таке план реалізації проєкту?
2. Які етапи включає програмний проєкт?
3. Що таке декомпозиція задач?
4. Що таке трудомісткість?
5. Які методи оцінювання трудомісткості існують?
6. Чому важливо враховувати залежності між задачами?
7. Яка роль ризиків у плануванні?
8. Чому планування є динамічним процесом?
9. Як оцінити реалістичність плану?
10. Які переваги правильного планування?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис проєкту;
– структура задач;
– оцінка трудомісткості;
– план реалізації;
– аналіз ризиків;
– висновки.

Практична робота 20. Оформлення документації та підготовка релізу програмного продукту

Мета роботи – ознайомитися з принципами підготовки супровідної документації програмного забезпечення, навчитися оформлювати основні документи та виконувати підготовку програмного продукту до релізу.

Ключові положення

Оформлення документації є завершальним етапом розробки програмного продукту, що забезпечує його зрозумілість, можливість використання, супроводження та подальшого розвитку. Документація містить відомості про функціональні можливості системи, особливості її використання, вимоги до середовища виконання та принципи роботи основних компонентів. Якісно підготовлена документація є необхідною умовою ефективної експлуатації програмного забезпечення.

Документація програмного продукту може включати різні види матеріалів. До них належать користувацька документація, технічна документація, інструкції з встановлення, опис інтерфейсів, специфікації, а також документи, що відображають процес розробки та тестування. Користувацька документація орієнтована на кінцевого користувача і пояснює, як працювати з програмою. Технічна документація призначена для розробників і містить відомості про структуру системи, її компоненти та механізми взаємодії.

Одним із важливих аспектів є структурованість документації. Вона повинна мати логічну побудову, що дозволяє швидко знаходити необхідну інформацію. Зазвичай документація включає вступ, опис призначення системи, вимоги до середовища, інструкції з встановлення, опис функцій, приклади використання та розділ із відповідями на типові питання або описом можливих проблем.

Підготовка релізу програмного продукту передбачає завершення розробки, перевірку працездатності системи та формування кінцевої версії, готової до використання. Реліз є офіційним випуском програмного продукту, що може бути переданий користувачам або замовнику. Перед випуском необхідно переконатися у стабільності роботи системи та відсутності критичних дефектів.

Важливим елементом релізу є формування версії програмного продукту. Нумерація версій дозволяє відстежувати розвиток системи, фіксувати внесені зміни та забезпечувати контроль за оновленнями. Кожна версія повинна супроводжуватися описом змін, що дає змогу користувачам зрозуміти, які нові можливості або виправлення були внесені.

Під час підготовки релізу необхідно виконати збірку програмного продукту. Це включає компіляцію, об’єднання модулів, підключення необхідних бібліотек та формування виконуваного файлу або пакета. Важливо забезпечити коректність збірки та перевірити її працездатність у цільовому середовищі.

Окрему увагу слід приділити тестуванню перед релізом. Необхідно переконатися, що всі основні функції працюють коректно, а критичні дефекти відсутні. Це може включати повторне виконання модульного, інтеграційного та системного тестування.

Важливим аспектом є підготовка інструкції з встановлення та використання програмного продукту. Така інструкція повинна містити вимоги до середовища, послідовність дій для встановлення, опис запуску програми та основні сценарії використання. Це забезпечує можливість самостійного використання продукту користувачами.

Документування змін є обов’язковою складовою релізу. Необхідно підготувати опис нової версії, що включає перелік доданих функцій, виправлених помилок та відомих обмежень. Такий документ дозволяє користувачам і розробникам орієнтуватися у розвитку системи.

Підготовка релізу також передбачає організацію структури файлів. Програмний продукт повинен бути упакований таким чином, щоб користувач міг легко встановити та

запустити його. До складу релізу можуть входити виконувані файли, бібліотеки, конфігураційні файли, документація та допоміжні матеріали.

Додатково слід зазначити, що процес підготовки релізу повинен бути відтворюваним. Це означає, що у разі необхідності має бути можливість повторно сформувати реліз на основі вихідного коду та налаштувань. Такий підхід забезпечує надійність і контроль за процесом розробки.

Також важливо враховувати, що реліз є точкою фіксації стану програмного продукту. Після випуску версії можуть виявлятися нові дефекти або виникати потреба у розширенні функціональності, що призводить до підготовки наступних релізів. Тому процес оформлення документації та підготовки релізу має бути впорядкованим і повторюваним.

Таким чином, оформлення документації та підготовка релізу є завершальними етапами створення програмного продукту, що забезпечують його готовність до використання, зрозумілість для користувачів та можливість подальшого розвитку.

Практичне завдання

1. Ознайомитися з видами документації програмного забезпечення.
2. Обрати програмний продукт для підготовки релізу.
3. Підготувати опис призначення системи.
4. Оформити користувацьку документацію.
5. Оформити технічну документацію.
6. Підготувати інструкцію з встановлення.
7. Сформувати опис версії програмного продукту.
8. Виконати збірку програмного продукту.
9. Перевірити працездатність релізної версії.
10. Сформувати структуру релізного пакета.
11. Зробити висновки щодо готовності продукту до використання.

Методичні вказівки до виконання практичного завдання

Перед початком виконання роботи необхідно ознайомитися з принципами документування програмного забезпечення та роллю документації у процесі його використання. Доцільно звернути увагу на те, що документація повинна бути зрозумілою, структурованою та відповідати призначенню програмного продукту.

На першому етапі необхідно обрати програмний продукт, що був розроблений у попередніх роботах. Це дозволить підготувати документацію для реального прикладу та завершити розробку системи.

Далі необхідно підготувати опис призначення системи. У цьому описі слід зазначити, для чого призначений програмний продукт, які задачі він вирішує та хто є його користувачем.

На наступному етапі потрібно оформити користувацьку документацію. Вона повинна містити опис інтерфейсу, основних функцій та приклади використання системи.

Після цього необхідно оформити технічну документацію. У ній слід описати структуру системи, її компоненти та особливості реалізації.

Далі необхідно підготувати інструкцію з встановлення. Слід описати вимоги до середовища та послідовність дій для встановлення програми.

На наступному етапі потрібно сформувати опис версії програмного продукту. У ньому слід зазначити номер версії та основні зміни.

Після цього необхідно виконати збірку програмного продукту та перевірити його працездатність.

Далі необхідно сформувати структуру релізного пакета, що включає програму, документацію та необхідні файли.

Після завершення роботи необхідно оцінити готовність продукту до використання та зробити висновки.

Таким чином, виконання роботи дозволяє сформувати навички оформлення документації та підготовки програмного продукту до релізу.

Контрольні питання

1. Що таке документація програмного забезпечення?
2. Які види документації існують?
3. Яке призначення користувацької документації?
4. Яке призначення технічної документації?
5. Що таке реліз програмного продукту?
6. Яка роль нумерації версій?
7. Що входить до складу релізного пакета?
8. Чому важливе тестування перед релізом?
9. Що таке інструкція з встановлення?
10. Чому важливо документувати зміни?

Зміст звіту

– назва практичної роботи;
– мета виконання;
– опис програмного продукту;
– користувацька документація;
– технічна документація;
– інструкція з встановлення;
– опис версії;
– структура релізного пакета;
– висновки.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Основна

1. Pressman, R. S., Maxim, B. R. Software Engineering: A Practitioners Approach. McGraw-Hill Education. 2019. 1408p. ISBN: 9781260548006, 1260548007.
2. Sommerville, I. Software Engineering, Global Edition. Pearson Education. 2016. 816p. ISBN: 9781292096148, 1292096144.
3. Martin, R.C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall. 2018. 404p. ISBN: 9780134494166, 0134494164.
4. Liang, Q. Continuous Delivery 2.0: Business-leading DevOps Essentials. CRC Press. 2021. 362p. ISBN: 9781000474763, 1000474763.
5. Chacon, Scott. Pro Git: Everything You Need to Know About Git. Apress, 2022. 497p.

Допоміжна

6. Ahmed, Ashfaque, and Prasad, Bhanu. Foundations of Software Engineering. CRC Press, 2016. 475p. ISBN: 9781498737609, 1498737609
7. Adewole, Ayobami. C# and .NET Core Test-Driven Development: Dive Into TDD to Create Flexible, Maintainable, and Production-ready .NET Core Applications. Packt Publishing, 2018. 300p. ISBN: 9781788299923, 1788299922
8. Peters, Lawrence J.. Software Project Management: Methods and Techniques. CRC Press. 244p. ISBN: 9781040035238, 104003523X.
9. Gechman, Marvin. Project Management of Large Software-Intensive Systems. CRC Press, 2019. 392p. ISBN: 9780429648168, 0429648162.
10. Singh, Shweta. Role of Project Management Software in the Manufacturing Sector. Германия, GRIN Verlag, 2022. 102p. ISBN: 9783346776754, 3346776751
11. ISO/IEC 25010:2011 Systems and software engineering – Software product quality requirements and evaluation (SQuaRE) – System and software quality models.
12. ISO/IEC/IEEE 12207:2017 Systems and software engineering – Software life cycle processes.

Інформаційні ресурси

13. Scrum Guide. URL: https://scrumguides.org
14. Git Documentation. URL: https://git-scm.com/docs
15. GitHub Docs. URL: https://docs.github.com
16. Atlassian Agile Coach. URL: https://www.atlassian.com/agile
17. Software Engineering Institute (SEI). URL: https://www.sei.cmu.edu

Навчальне видання

Лебедєва Олена Юріївна

ТЕХНОЛОГІЇ СТВОРЕННЯ ПРОГРАМНИХ ПРОДУКТІВ

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

Українською мовою