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

О.Ю. Лебедєва

ТЕХНОЛОГІЇ СТВОРЕННЯ ПРОГРАМНИХ ПРОДУКТІВ

Опорний конспект лекцій
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. Опорний конспект лекцій для здобувачів вищої освіти, які навчаються за спеціальностями галузі «Інформаційні технології» [Електронне видання]. Кафедра інформаційних технологій Міжнародного гуманітарного університету. Одеса, 2025. 33 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

Лекція 1. Поняття програмного продукту та життєвий цикл програмного забезпечення

Поняття програмного продукту є базовим у дисципліні Технології створення програмних продуктів, оскільки саме воно визначає об'єкт розробки, його структуру, характеристики та вимоги до якості. У практиці інженерії програмного забезпечення важливо чітко розрізняти програмний код як технічний результат діяльності розробника та програмний продукт як комплексне рішення, орієнтоване на кінцевого користувача або замовника.

Програмний код є сукупністю інструкцій, записаних мовою програмування, що призначені для виконання комп'ютерною системою. Він є лише частиною програмного продукту і відображає алгоритмічну та логічну складову програмної системи. Програмний код може існувати як окремий артефакт, наприклад у навчальних або експериментальних проєктах, однак у більшості випадків він не є завершеним продуктом з точки зору практичного використання.

Програмний продукт, на відміну від програмного коду, є комплексним результатом інженерної діяльності, який містить не лише виконуваний код, але й супровідну документацію, інтерфейс користувача, засоби налаштування, механізми обробки помилок, систему оновлення, інсталяційні пакети, а також елементи забезпечення якості. До складу програмного продукту входять також тестові сценарії, результати тестування, інструкції для користувачів та адміністраторів, а також інші компоненти, необхідні для повноцінного використання системи.

Ключовою відмінністю програмного продукту є його орієнтація на кінцевого користувача та відповідність визначеним вимогам. Програмний продукт створюється з урахуванням функціональних та нефункціональних вимог, включаючи продуктивність, надійність, безпеку, зручність використання та можливість масштабування. Таким чином, програмний продукт є результатом системного підходу до розробки, що передбачає планування, аналіз, проєктування, реалізацію, тестування та супровід.

Життєвий цикл програмного забезпечення є концепцією, що описує сукупність процесів, які відбуваються від моменту формування ідеї програмного продукту до завершення його експлуатації. Життєвий цикл охоплює всі етапи створення, впровадження, використання та виведення програмного продукту з експлуатації. Він є основою для організації процесу розробки та управління проєктом.

Основними процесами життєвого циклу є процеси розробки, підтримки та управління. Процес розробки містить діяльність, пов'язану з аналізом вимог, проєктуванням системи, реалізацією програмного коду, тестуванням та інтеграцією компонентів. Процес підтримки охоплює діяльність, спрямовану на забезпечення працездатності програмного продукту після його впровадження, включаючи виправлення помилок, оновлення та адаптацію до змін у середовищі. Процес управління містить планування, контроль виконання, управління ризиками та ресурсами.

Життєвий цикл програмного забезпечення традиційно поділяється на етапи, серед яких можна виділити аналіз вимог, проєктування, реалізацію, тестування, впровадження та супровід. На етапі аналізу вимог виконується визначення функціональних можливостей

системи, формуються вимоги замовника та користувачів. На етапі проєктування створюється архітектура системи, визначаються компоненти, їх взаємодія та структура даних. Етап реалізації передбачає безпосереднє написання програмного коду відповідно до розробленого проєкту. Тестування забезпечує перевірку відповідності системи вимогам та виявлення дефектів. Впровадження містить встановлення системи у робочому середовищі, а супровід забезпечує її подальше функціонування.

У сучасній практиці розробки програмного забезпечення важливу роль відіграє ітераційний підхід. Ітерація є повторюваним циклом виконання етапів життєвого циклу, у межах якого створюється частина функціональності системи. Такий підхід дає змогу поступово розвивати програмний продукт, отримувати зворотний зв'язок від користувачів та вносити зміни на ранніх етапах розробки.

Існує декілька моделей життєвого циклу програмного забезпечення, що визначають порядок виконання етапів та характер взаємодії між ними. Найбільш поширеними є каскадна, інкрементна та спіральна моделі.

Каскадна модель є однією з найстаріших та найпростіших моделей організації процесу розробки. Вона передбачає послідовне виконання етапів життєвого циклу, де кожен наступний етап починається лише після завершення попереднього. Основними характеристиками каскадної моделі є чітка структурованість, документованість та контрольованість процесу. Кожен етап має визначені результати, які використовуються на наступних етапах.

Перевагами каскадної моделі є простота управління, зрозумілість структури процесу, можливість чіткої фіксації вимог та планування ресурсів. Вона добре підходить для проєктів з чітко визначеними та стабільними вимогами, де зміни у процесі розробки є малоймовірними. Однак каскадна модель має суттєві обмеження. Основним недоліком є низька гнучкість, оскільки внесення змін на пізніх етапах є складним та дорогим. Крім того, користувач отримує робочий продукт лише наприкінці розробки, що ускладнює врахування зворотного зв'язку.

Інкрементна модель передбачає розробку програмного продукту у вигляді послідовності інкрементів, кожен з яких додає нову функціональність до системи. На відміну від каскадної моделі, інкрементна модель дає змогу отримувати частково працездатний продукт на ранніх етапах розробки. Кожен інкремент проходить усі етапи життєвого циклу, включаючи аналіз, проєктування, реалізацію та тестування.

Перевагами інкрементної моделі є можливість раннього отримання результатів, підвищення гнучкості та адаптивності до змін вимог, а також зниження ризиків. Замовник може оцінювати проміжні результати та вносити корективи у процесі розробки. До обмежень інкрементної моделі належить необхідність ретельного планування структури інкрементів та забезпечення узгодженості між ними. Крім того, складність управління може зростати зі збільшенням кількості інкрементів.

Спіральна модель поєднує елементи ітераційного підходу та управління ризиками. Вона передбачає виконання розробки у вигляді послідовності витків спіралі, кожен з яких містить етапи планування, аналізу ризиків, розробки та оцінювання результатів. Особливістю спіральної моделі є акцент на ідентифікації та мінімізації ризиків на кожному етапі розробки.

Перевагами спіральної моделі є високий рівень гнучкості, можливість раннього виявлення проблем та адаптації до змін. Вона добре підходить для складних та ризикованих проєктів, де вимоги можуть змінюватися у процесі розробки. Однак спіральна модель є

складною у реалізації та потребує значних ресурсів для управління ризиками. Вона також вимагає високої кваліфікації команди та чіткої організації процесів.

Вибір моделі життєвого циклу програмного забезпечення залежить від багатьох факторів, включаючи тип проєкту, рівень визначеності вимог, складність системи, доступні ресурси та часові обмеження. Для невеликих проєктів із чітко визначеними вимогами доцільно використовувати каскадну модель, оскільки вона забезпечує простоту та передбачуваність процесу. Для проєктів, де вимоги можуть змінюватися, більш ефективною є інкрементна модель, що дає змогу поступово розвивати функціональність системи.

У випадку складних систем із високим рівнем невизначеності та ризиків доцільно застосовувати спіральну модель, яка забезпечує систематичний підхід до управління ризиками. Вибір моделі також може залежати від досвіду команди, організаційних стандартів та вимог замовника.

Таким чином, розуміння відмінностей між програмним кодом і програмним продуктом, а також знання моделей життєвого циклу програмного забезпечення є необхідною умовою для ефективної організації процесу розробки. Вибір адекватної моделі дозволяє оптимізувати використання ресурсів, знизити ризики та забезпечити відповідність програмного продукту вимогам користувачів і замовників.

Лекція 2. Гнучкі методології розробки програмних продуктів

У сучасній інженерії програмного забезпечення дедалі більшого поширення набувають гнучкі методології розробки, що орієнтовані на адаптивність, швидке реагування на зміни та тісну взаємодію з замовником. На відміну від традиційних підходів, які передбачають жорстку послідовність етапів, гнучкі методології базуються на ітераційному розвитку програмного продукту та постійному уточненні вимог у процесі роботи.

Гнучкі методології об’єднуються загальною ідеологією, що сформульована в Agile Manifesto. Цей документ визначає ключові цінності та принципи, які лежать в основі сучасних підходів до організації розробки програмного забезпечення. Основна увага приділяється людям та їх взаємодії, робочому програмному продукту, співпраці із замовником та готовності до змін.

Згідно з принципами Agile, процес розробки повинен бути орієнтований на постачання працездатного програмного продукту невеликими ітераціями. Кожна ітерація завершується отриманням результату, який може бути оцінений замовником. Це дає змогу своєчасно виявляти помилки, уточнювати вимоги та коригувати напрям розвитку системи. Важливим є також принцип постійної комунікації між учасниками команди, що забезпечує узгодженість рішень та зменшує ймовірність непорозумінь.

Однією з найпоширеніших реалізацій Agile є методологія Scrum. Вона визначає чітку структуру ролей, артефактів та подій, що забезпечують організацію командної роботи у процесі розробки.

У Scrum виділяють три основні ролі. Власник продукту відповідає за формування та пріоритизацію вимог до програмного продукту. Він визначає, які функції мають бути реалізовані в першу чергу, та підтримує актуальність списку завдань. Scrum-майстер забезпечує дотримання принципів і правил Scrum, усуває перешкоди у роботі команди та сприяє ефективній взаємодії між її учасниками. Команда розробки є самокерованою групою

фахівців, які безпосередньо виконують роботи з аналізу, проєктування, реалізації та тестування програмного продукту.

До основних артефактів Scrum належить беклог продукту, що містить перелік усіх вимог та задач, які потрібно реалізувати. Кожен елемент беклогу описує окрему функціональність або вимогу до системи. Беклог спринту є підмножиною беклогу продукту, яка обрана для реалізації в межах конкретної ітерації. Інкремент є результатом роботи команди за спринт та представляє собою готову до використання частину програмного продукту.

Події Scrum визначають ритм роботи команди. Спринт є фіксованим за часом інтервалом, протягом якого створюється інкремент продукту. Планування спринту передбачає визначення обсягу робіт та постановку цілей. Щоденні зустрічі дають змогу координувати діяльність команди та швидко вирішувати поточні питання. Огляд спринту використовується для демонстрації результатів замовнику та отримання зворотного зв'язку. Ретроспектива дозволяє проаналізувати процес роботи та визначити шляхи його покращення.

Іншим підходом до організації гнучкої розробки є Kanban, який орієнтований на управління потоком задач. Основною ідеєю Kanban є візуалізація процесу роботи та обмеження кількості задач, що виконуються одночасно. Для цього використовується дошка, на якій задачі розміщуються відповідно до їх стану, наприклад «заплановано», «в роботі», «завершено».

Важливою особливістю Kanban є обмеження кількості задач у кожному стані, що дає змогу уникнути перевантаження команди та забезпечити рівномірний потік роботи. Задачі переміщуються між станами у міру їх виконання, що забезпечує прозорість процесу та дає змогу швидко виявляти вузькі місця. На відміну від Scrum, Kanban не передбачає жорстко визначених ітерацій, що робить його більш гнучким у застосуванні.

Методологія Extreme Programming є ще одним прикладом гнучкого підходу, що зосереджений на підвищенні якості програмного забезпечення та ефективності розробки. XP містить набір практик, спрямованих на забезпечення надійності та підтримуваності програмного коду.

Серед основних практик XP можна виділити парне програмування, при якому два розробники працюють разом над одним завданням. Один виконує написання коду, інший здійснює контроль та аналіз, що дає змогу підвищити якість результату. Тестування є невід'ємною частиною процесу, причому широко використовується підхід розробки через тестування, коли спочатку створюються тести, а потім реалізується код, що їх проходить.

Безперервна інтеграція передбачає регулярне об'єднання змін у спільному репозиторії та автоматичне виконання тестів. Це дозволяє швидко виявляти помилки та забезпечує стабільність системи. Рефакторинг є процесом покращення структури коду без зміни його функціональності, що сприяє підтримуваності та розширюваності програмного продукту.

Організація ітераційної роботи є ключовим елементом гнучких методологій. Ітерація являє собою обмежений у часі цикл, протягом якого виконується визначений обсяг робіт. На початку ітерації формується план, визначаються задачі та розподіляються ресурси. У процесі виконання здійснюється постійний контроль прогресу, а наприкінці проводиться оцінювання результатів.

Ітераційний підхід дозволяє зменшити ризики, пов'язані з невизначеністю вимог, оскільки кожна ітерація забезпечує отримання зворотного зв'язку та можливість коригування планів. Крім того, він сприяє більш ефективному використанню ресурсів та підвищенню якості програмного продукту.

Комунікація відіграє визначальну роль у процесі розробки програмного забезпечення. У гнучких методологіях особлива увага приділяється безпосередній взаємодії між учасниками команди, а також між командою та замовником. Регулярні зустрічі, обговорення та демонстрації результатів дозволяють забезпечити узгодженість дій та своєчасне вирішення проблем.

Ефективна комунікація передбачає чітке формулювання вимог, прозорість процесів та відкритість до обговорення. Використання інструментів спільної роботи, систем контролю версій та засобів відстеження задач дає змогу забезпечити доступність інформації для всіх учасників процесу. Важливим є також документування ключових рішень, що дозволяє уникнути втрати інформації та забезпечити спадковість знань.

Узгодження рішень у процесі розробки здійснюється на основі колективного обговорення та аналізу альтернатив. Команда повинна враховувати як технічні аспекти, так і вимоги замовника, що забезпечує баланс між якістю програмного продукту та його відповідністю потребам користувачів. У гнучких методологіях важливу роль відіграє самоорганізація команди, що дає змогу підвищити ефективність прийняття рішень та відповідальність за результат.

Таким чином, гнучкі методології розробки програмних продуктів забезпечують ефективну організацію процесу створення програмного забезпечення в умовах невизначеності та змін. Використання принципів Agile, застосування Scrum, Kanban та XP-практик, а також належна організація ітераційної роботи та комунікації дозволяють підвищити якість програмного продукту, скоротити терміни розробки та забезпечити задоволення вимог замовника.

Лекція 3. Формування та аналіз вимог до програмного забезпечення

Формування та аналіз вимог є одним із ключових етапів створення програмного продукту, оскільки саме на цьому етапі визначається, що саме повинна виконувати програмна система та яким вимогам вона має відповідати. Якість визначення вимог безпосередньо впливає на успішність усього проєкту, оскільки помилки, допущені на цьому етапі, призводять до значних витрат на їх виправлення у подальшому.

Вимоги до програмного забезпечення являють собою формалізований опис функціональних можливостей та обмежень програмного продукту. Вони визначають очікувану поведінку системи, її характеристики та умови експлуатації. Формування вимог передбачає їх виявлення, аналіз, документування та узгодження з усіма зацікавленими сторонами.

Джерела вимог є різноманітними та залежать від типу програмного продукту, предметної області та умов його використання. Основним джерелом є замовник, який формулює загальні очікування щодо функціональності системи. Кінцеві користувачі також відіграють важливу роль, оскільки вони безпосередньо взаємодіють із програмним продуктом і можуть надати інформацію про свої потреби та очікування.

До інших джерел вимог належать нормативні документи, стандарти, технічні регламенти, а також існуючі програмні системи, що виконують аналогічні функції. Аналіз предметної області дозволяє визначити бізнес-процеси, які мають бути автоматизовані, та сформувати відповідні вимоги. Важливим джерелом є також результати попередніх проєктів, що дозволяють врахувати накопичений досвід та типові рішення.

Функціональні вимоги визначають, які саме функції повинна виконувати система. Вони описують поведінку програмного продукту у відповідь на дії користувача або інші події. Наприклад, функціональні вимоги можуть визначати обробку запитів, виконання обчислень, збереження даних або взаємодію з іншими системами. Такі вимоги зазвичай формулюються у вигляді сценаріїв використання або опису функціональних можливостей.

Нефункціональні вимоги визначають характеристики якості програмного продукту та обмеження, що накладаються на його реалізацію. До них належать вимоги до продуктивності, надійності, безпеки, масштабованості, зручності використання та сумісності. Нефункціональні вимоги не описують конкретну поведінку системи, але визначають умови, за яких вона повинна функціонувати. Наприклад, вимога до часу відповіді системи або до рівня доступності сервісу.

Специфікація вимог до програмного забезпечення є документом, що містить повний і формалізований опис вимог до системи. Вона використовується як основа для подальших етапів розробки, включаючи проєктування, реалізацію та тестування. Специфікація повинна бути чіткою, однозначною, повною та узгодженою.

У процесі створення специфікації важливо забезпечити структурованість інформації. Документ зазвичай містить опис загальних характеристик системи, функціональні вимоги, нефункціональні вимоги, обмеження, а також інтерфейси взаємодії. Важливим є також використання єдиної термінології та формалізованих описів, що зменшує ризик неправильного трактування вимог.

Трасування вимог є процесом встановлення зв’язків між вимогами та іншими артефактами розробки. Це дозволяє відслідковувати, як кожна вимога реалізується у програмному продукті, а також перевіряти її відповідність тестам, проєктним рішенням та реалізованому коду. Трасування забезпечує прозорість процесу розробки та полегшує управління змінами.

Існує декілька видів трасування, включаючи пряме та зворотне. Пряме трасування дозволяє визначити, які елементи системи реалізують конкретну вимогу. Зворотне трасування дає змогу перевірити, які вимоги відповідають певним компонентам системи. Використання трасування є особливо важливим у складних проєктах, де кількість вимог є значною.

Управління змінами вимог є невід’ємною частиною процесу розробки програмного забезпечення. У реальних умовах вимоги рідко залишаються незмінними, оскільки можуть змінюватися потреби замовника, умови ринку або технічні обмеження. Тому необхідно забезпечити контрольований процес внесення змін.

Процес управління змінами включає реєстрацію запитів на зміну, їх аналіз, оцінку впливу на проєкт, прийняття рішення щодо впровадження та оновлення документації. Важливим є визначення відповідальних осіб та процедур, що забезпечують узгодженість змін із загальною стратегією розробки.

Оцінка впливу змін дозволяє визначити, які компоненти системи будуть затронуті, які ресурси необхідні для реалізації змін та як це вплине на терміни виконання проєкту. Це дає змогу приймати обґрунтовані рішення та мінімізувати ризики.

Управління змінами також передбачає підтримку актуальності специфікації вимог. Кожна зміна повинна бути відображена у документації, що забезпечує її відповідність реальному стану системи. Використання систем керування версіями дозволяє відслідковувати історію змін та забезпечує можливість повернення до попередніх версій.

Важливою складовою процесу є узгодження змін із зацікавленими сторонами. Це забезпечує відповідність змін очікуванням замовника та дозволяє уникнути конфліктів. Регулярна комунікація між учасниками проєкту сприяє своєчасному виявленню необхідності змін та їх ефективному впровадженню.

Таким чином, формування та аналіз вимог є критично важливим етапом створення програмного продукту, що визначає його функціональність та якість. Використання структурованих підходів до збору, аналізу та документування вимог, застосування трасування та ефективне управління змінами дозволяють забезпечити успішну реалізацію проєкту та відповідність програмного продукту потребам користувачів і замовників.

ТЕМА 2. ПРОЄКТУВАННЯ ТА АРХІТЕКТУРА ПРОГРАМНИХ СИСТЕМ

Лекція 4. Архітектура програмного забезпечення та принципи побудови програмних систем

Архітектура програмного забезпечення є одним із ключових аспектів створення програмного продукту, оскільки визначає загальну структуру системи, спосіб взаємодії її компонентів та принципи організації програмного коду. Вона формує основу для подальшої реалізації, тестування та супроводу системи, а також безпосередньо впливає на її якісні характеристики, зокрема масштабованість, надійність та розширюваність.

Під архітектурою програмного забезпечення розуміють сукупність структурних елементів системи, їх взаємозв'язків, а також принципів і правил, за якими здійснюється побудова та розвиток програмного продукту. Архітектура визначає, як саме система буде розподілена на компоненти, які функції виконуватиме кожен з них, а також яким чином ці компоненти взаємодіють між собою.

Архітектурні рішення приймаються на ранніх етапах розробки і мають довготривалий вплив на життєвий цикл програмного забезпечення. Неправильно обрана архітектура може призвести до ускладнення розробки, зростання витрат на підтримку та обмеження можливостей розвитку системи. Тому важливим є врахування вимог до системи, її призначення, очікуваного навантаження та умов експлуатації.

Серед основних архітектурних стилів, що застосовуються у сучасній розробці програмного забезпечення, можна виділити монолітну, клієнт-серверну, багаторівневу та мікросервісну архітектури.

Монолітна архітектура передбачає створення програмного продукту у вигляді єдиного виконуваного модуля, в якому всі функціональні компоненти об'єднані в одну систему. У такій архітектурі логіка обробки даних, інтерфейс користувача та доступ до даних реалізуються в межах одного програмного застосунку.

Перевагою монолітної архітектури є відносна простота реалізації, оскільки всі компоненти знаходяться в одному середовищі виконання. Це спрощує налагодження, тестування та розгортання системи. Крім того, взаємодія між компонентами здійснюється без додаткових витрат на мережеву комунікацію.

Однак монолітна архітектура має суттєві обмеження. Зі зростанням розміру системи вона стає складною у підтримці та модифікації. Внесення змін в один компонент може впливати на інші частини системи, що ускладнює розвиток програмного продукту. Масштабування такої системи зазвичай виконується шляхом збільшення ресурсів, що не завжди є ефективним.

Клієнт-серверна архітектура передбачає розподіл системи на дві основні частини: клієнтську та серверну. Клієнт відповідає за взаємодію з користувачем, тоді як сервер виконує обробку даних, бізнес-логіку та зберігання інформації. Взаємодія між клієнтом і сервером здійснюється через мережеві протоколи.

Перевагою клієнт-серверної архітектури є розподіл відповідальності між компонентами, що дозволяє спростити розробку та підтримку системи. Вона забезпечує централізоване управління даними та підвищує рівень безпеки. Крім того, така архітектура дозволяє використовувати різні типи клієнтів, наприклад вебзастосунки або мобільні застосунки.

До недоліків належить залежність від серверної частини, що може стати вузьким місцем у системі. При великій кількості користувачів сервер може перевантажуватися, що потребує додаткових механізмів масштабування.

Багаторівнева архітектура є розвитком клієнт-серверного підходу та передбачає поділ системи на декілька рівнів, кожен з яких виконує окрему функцію. Зазвичай виділяють рівень представлення, рівень бізнес-логіки та рівень доступу до даних. Такий підхід дає змогу чітко розмежувати відповідальність між компонентами системи.

Перевагою багаторівневої архітектури є підвищення модульності та гнучкості системи. Кожен рівень може розроблятися та модифікуватися незалежно від інших, що спрощує супровід та розвиток програмного продукту. Крім того, така архітектура забезпечує можливість масштабування окремих рівнів залежно від навантаження.

До обмежень належить ускладнення структури системи та необхідність організації взаємодії між рівнями. Це може призводити до збільшення складності реалізації та зростання витрат на розробку.

Мікросервісна архітектура передбачає побудову системи у вигляді набору незалежних сервісів, кожен з яких виконує окрему функцію та взаємодіє з іншими через визначені інтерфейси. Кожен сервіс може розроблятися, розгортатися та масштабуватися незалежно від інших.

Перевагами мікросервісної архітектури є висока гнучкість, можливість незалежного розвитку компонентів та ефективне масштабування. Вона дозволяє використовувати різні технології для реалізації окремих сервісів, що підвищує адаптивність системи до змін. Крім того, відмова одного сервісу не обов’язково призводить до зупинки всієї системи.

Разом з тим, мікросервісна архітектура є складною у реалізації та потребує значних ресурсів для організації взаємодії між сервісами. Необхідно забезпечити надійну мережеву комунікацію, управління конфігураціями та моніторинг системи. Крім того, виникають додаткові складності, пов’язані з узгодженістю даних між сервісами.

Принципи модульності та декомпозиції є фундаментальними для побудови ефективної архітектури програмного забезпечення. Модульність передбачає розподіл системи на окремі компоненти, кожен з яких виконує визначену функцію. Це дозволяє зменшити складність системи, підвищити її зрозумілість та спростити тестування.

Декомпозиція є процесом поділу складної системи на більш прості частини. Вона дозволяє визначити структуру системи, виділити окремі підсистеми та встановити взаємозв’язки між ними. Ефективна декомпозиція забезпечує можливість незалежної розробки та модифікації компонентів.

Важливими характеристиками модульної системи є слабка зв’язаність та висока згуртованість. Слабка зв’язаність означає мінімальну залежність між модулями, що дозволяє змінювати один модуль без впливу на інші. Висока згуртованість передбачає, що всі елементи модуля спрямовані на виконання однієї функції.

Забезпечення масштабованості програмної системи є важливим завданням при проєктуванні архітектури. Масштабованість означає здатність системи ефективно працювати при зростанні навантаження. Існують два основні підходи до масштабування: вертикальне та горизонтальне. Вертикальне масштабування передбачає збільшення ресурсів одного вузла, тоді як горизонтальне масштабування полягає у додаванні нових вузлів до системи.

Для забезпечення масштабованості використовуються різні архітектурні рішення, зокрема балансування навантаження, розподіл функціональності між компонентами та

використання кешування. Вибір підходу залежить від типу системи та вимог до її продуктивності.

Розширюваність системи визначає можливість додавання нової функціональності без значних змін у вже існуючому коді. Це досягається за рахунок використання модульної структури, чітко визначених інтерфейсів та принципів абстракції. Архітектура повинна передбачати можливість розвитку системи у майбутньому, що є особливо важливим для довготривалих проєктів.

Таким чином, архітектура програмного забезпечення є визначальним фактором, що впливає на ефективність розробки та експлуатації програмного продукту. Використання відповідних архітектурних стилів, застосування принципів модульності та декомпозиції, а також забезпечення масштабованості та розширюваності дозволяють створювати ефективні, надійні та адаптивні програмні системи.

Лекція 5. Моделювання структури та поведінки програмних систем. UML та проєктування інтерфейсів

Моделювання є важливим етапом створення програмного продукту, що дає змогу формалізувати уявлення про систему, її структуру, поведінку та взаємодію компонентів до початку реалізації. Використання моделей дозволяє зменшити складність розробки, виявити потенційні проблеми на ранніх етапах та забезпечити узгодженість рішень між учасниками проєкту.

Модель програмної системи є абстрактним представленням її властивостей, що дозволяє описати ключові аспекти функціонування без деталізації реалізації. Моделювання структури передбачає опис компонентів системи та їх взаємозв’язків, тоді як моделювання поведінки визначає, як система реагує на події та взаємодіє з користувачами або іншими системами.

Одним із найбільш поширених інструментів моделювання є Unified Modeling Language, що являє собою стандартизовану мову графічного опису програмних систем. UML дозволяє створювати різні типи діаграм, які відображають структуру та поведінку системи з різних точок зору.

Діаграми варіантів використання застосовуються для опису функціональних можливостей системи з точки зору користувача. Вони відображають взаємодію між акторами та системою, визначаючи сценарії використання. Актори представляють зовнішні сутності, які взаємодіють із системою, наприклад користувачів або інші програмні системи. Варіанти використання описують функції, які система надає акторам.

Такі діаграми дозволяють визначити межі системи, її функціональність та основні сценарії роботи. Вони є важливим інструментом на етапі формування вимог, оскільки забезпечують зрозуміле представлення функціональних можливостей системи для замовника.

Діаграми класів використовуються для моделювання структури програмної системи. Вони відображають класи, їх атрибути, методи та зв’язки між класами. Зв’язки можуть включати асоціацію, агрегацію, композицію та успадкування. Діаграми класів дозволяють визначити внутрішню структуру системи та організацію програмного коду.

Такі діаграми є основою для реалізації об’єктно-орієнтованих систем, оскільки вони відображають концептуальну модель предметної області. Вони також використовуються для аналізу залежностей між компонентами та оптимізації структури системи.

Діаграми послідовностей застосовуються для моделювання поведінки системи у часі. Вони відображають взаємодію між об’єктами у вигляді обміну повідомленнями. Кожен об’єкт представлений вертикальною лінією, а повідомлення відображаються у вигляді стрілок між об’єктами.

Діаграми послідовностей дозволяють зрозуміти порядок виконання операцій, визначити взаємодію компонентів та виявити можливі проблеми у логіці системи. Вони є корисними при проєктуванні складних сценаріїв взаємодії та інтеграції компонентів.

Діаграми компонентів використовуються для опису високорівневої структури системи. Вони відображають основні компоненти програмного продукту та їх залежності. Компоненти можуть представляти модулі, бібліотеки або окремі сервіси.

Такі діаграми дозволяють визначити архітектуру системи, розподіл функціональності між компонентами та інтерфейси взаємодії. Вони є важливими для планування розгортання системи та організації командної роботи.

Проєктування інтерфейсів взаємодії є важливим аспектом створення програмного продукту, оскільки саме через інтерфейси здійснюється взаємодія між компонентами системи або між системою та зовнішніми користувачами. Інтерфейс визначає набір доступних операцій, формат даних та правила їх використання.

Інтерфейси повинні бути чітко визначеними, узгодженими та стабільними. Важливо забезпечити їх незалежність від внутрішньої реалізації компонентів, що дозволяє змінювати реалізацію без впливу на інші частини системи. Це досягається за рахунок використання принципів абстракції та інкапсуляції.

Контракти взаємодії визначають умови використання інтерфейсів, включаючи вхідні та вихідні параметри, передумови виконання операцій та очікувані результати. Контракт задає чіткі правила взаємодії між компонентами та дозволяє уникнути неоднозначностей.

У контрактах можуть визначатися обмеження на значення параметрів, формат даних, обробка помилок та поведінка системи у різних ситуаціях. Використання контрактів сприяє підвищенню надійності системи та полегшує інтеграцію компонентів.

Специфікації API є формалізованим описом інтерфейсів програмування застосунків, що визначають способи взаємодії між програмними компонентами або системами. Вони містять опис доступних методів, параметрів, форматів запитів і відповідей, а також можливих помилок.

API-специфікації можуть бути представлені у вигляді текстових описів або формалізованих документів, що використовуються для автоматизації процесів розробки та тестування. Вони є важливим інструментом при розробці розподілених систем та інтеграції різних компонентів.

Якісно розроблена специфікація API забезпечує узгодженість взаємодії між компонентами, зменшує ймовірність помилок та спрощує процес розробки. Вона також дозволяє незалежно розробляти різні частини системи, що підвищує ефективність командної роботи.

Моделювання структури та поведінки системи у поєднанні з проєктуванням інтерфейсів взаємодії дозволяє створити цілісне уявлення про програмний продукт ще до початку його реалізації. Це забезпечує зменшення ризиків, підвищення якості розробки та ефективну взаємодію між учасниками проєкту.

Таким чином, використання UML-діаграм для моделювання, а також застосування принципів проєктування інтерфейсів, контрактів та API-специфікацій є необхідною умовою для створення складних програмних систем, що відповідають сучасним вимогам до якості, масштабованості та підтримуваності.

Лекція 6. Компонентна розробка програмних продуктів та управління залежностями

Компонентна розробка є сучасним підходом до створення програмних продуктів, що базується на використанні незалежних програмних компонентів як основних одиниць побудови системи. Такий підхід дозволяє підвищити ефективність розробки, забезпечити повторне використання програмного коду та зменшити складність створення складних програмних систем.

Під програмним компонентом розуміють автономну частину програмного забезпечення, яка має чітко визначений інтерфейс і може використовуватися незалежно від інших компонентів. Компонент реалізує певну функціональність і взаємодіє з іншими частинами системи через стандартизовані інтерфейси. Важливою характеристикою компонента є його незалежність від внутрішньої реалізації інших компонентів, що забезпечує гнучкість системи.

Компонентна розробка передбачає створення програмного продукту шляхом поєднання окремих компонентів, які можуть бути розроблені як у межах одного проєкту, так і отримані з зовнішніх джерел. Це дозволяє скоротити час розробки, зменшити кількість помилок та підвищити якість програмного продукту.

Повторне використання модулів є одним із ключових принципів компонентного підходу. Воно полягає у застосуванні вже існуючих компонентів у нових програмних системах без необхідності їх повторної реалізації. Повторне використання дозволяє знизити витрати на розробку, підвищити надійність програмного забезпечення та забезпечити узгодженість функціональності.

Для ефективного повторного використання необхідно забезпечити відповідну організацію компонентів, зокрема їх документування, стандартизацію інтерфейсів та підтримку сумісності. Компоненти повинні бути достатньо універсальними, щоб їх можна було застосовувати у різних проєктах, але водночас спеціалізованими для виконання конкретних задач.

Інтеграція програмних компонентів є процесом об’єднання окремих компонентів у єдину систему. Вона передбачає налаштування взаємодії між компонентами, узгодження форматів даних та забезпечення коректної роботи всієї системи. Інтеграція може виконуватися поступово, у міру додавання нових компонентів, або одночасно для всієї системи.

Під час інтеграції важливо враховувати залежності між компонентами, оскільки вони визначають порядок їх підключення та взаємодії. Неправильне управління залежностями може призвести до конфліктів, помилок у виконанні та ускладнення супроводу системи.

Управління залежностями є процесом визначення, контролю та підтримки зв’язків між компонентами програмного продукту. Залежності виникають, коли один компонент використовує функціональність іншого. У складних системах кількість залежностей може бути знаною, що ускладнює процес розробки.

Для ефективного управління залежностями використовуються спеціалізовані інструменти та підходи, що дозволяють автоматизувати процес підключення компонентів, контролювати їх версії та забезпечувати узгодженість системи. Важливим є також мінімізація кількості залежностей та їх ізоляція, що дозволяє зменшити вплив змін у одному компоненті на інші.

Одним із принципів управління залежностями є інверсія залежностей, що передбачає залежність від абстракцій, а не від конкретних реалізацій. Це дозволяє підвищити гнучкість системи та спростити її модифікацію. Використання інтерфейсів та абстрактних класів дає змогу змінювати реалізацію компонентів без впливу на інші частини системи.

Версіонування компонентів є важливим аспектом управління залежностями. Воно передбачає присвоєння компонентам унікальних версій, що відображають стан їх розвитку. Версії дозволяють визначити, які зміни були внесені у компонент, та забезпечують можливість використання різних версій у різних проєктах.

Зазвичай використовується семантичне версіонування, яке передбачає поділ номера версії на три частини: основну, другорядну та номер виправлення. Основна версія змінюється при внесенні несумісних змін, другорядна – при додаванні нової функціональності без порушення сумісності, а номер виправлення – при усуненні помилок.

Сумісність компонентів визначає можливість їх спільного використання без виникнення конфліктів. Вона може бути зворотною, коли нова версія компонента сумісна зі старими, або прямою, коли старі версії можуть працювати з новими компонентами. Забезпечення сумісності є важливим для стабільності системи та спрощення її оновлення.

Порушення сумісності може призвести до необхідності внесення змін у залежні компоненти, що ускладнює процес супроводу. Тому при розробці компонентів важливо дотримуватися принципів стабільності інтерфейсів та передбачати можливість еволюції системи.

Для підтримки сумісності використовуються різні підходи, зокрема використання адаптерів, версіонування інтерфейсів та забезпечення зворотної сумісності. Це дозволяє поступово впроваджувати зміни без порушення роботи системи.

Компонентна розробка також передбачає використання репозиторіїв компонентів, де зберігаються програмні модулі та їх версії. Це дозволяє централізовано управляти компонентами, забезпечувати доступ до них та контролювати їх використання. Репозиторії можуть бути локальними або віддаленими, залежно від організації процесу розробки.

Таким чином, компонентна розробка є ефективним підходом до створення програмних продуктів, що дозволяє підвищити продуктивність розробки та якість системи. Використання принципів повторного використання, ефективної інтеграції, управління залежностями, а також забезпечення версіонування та сумісності компонентів дозволяє створювати гнучкі, масштабовані та підтримувані програмні системи.

ТЕМА 3. ТЕХНОЛОГІЇ РЕАЛІЗАЦІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Лекція 7. Організація роботи з використанням систем контролю версій

Системи контролю версій є невід'ємною складовою сучасного процесу розробки програмних продуктів. Вони забезпечують збереження історії змін програмного коду, підтримку колективної роботи розробників та контроль узгодженості змін у межах проєкту. Використання систем контролю версій дозволяє організувати процес розробки таким чином, щоб мінімізувати ризики втрати даних, конфліктів між змінами та помилок інтеграції.

Система контролю версій являє собою програмний інструмент, що дозволяє відстежувати зміни у файлах, зберігати попередні версії та забезпечувати можливість повернення до будь-якого стану проєкту. У сучасній практиці широко використовується розподілена система контролю версій Git, яка забезпечує гнучкість та високу ефективність у роботі з програмним кодом.

Основною одиницею організації даних у Git є репозиторій. Репозиторій містить усі файли проєкту, а також історію їх змін. Він може бути локальним, розташованим на комп'ютері розробника, або віддаленим, що знаходиться на сервері та використовується для спільної роботи. Наявність локального репозиторію дозволяє виконувати більшість операцій без підключення до мережі.

Зміни у репозиторії фіксуються за допомогою комітів. Коміт є логічно завершеною одиницею змін, що містить інформацію про внесені зміни, автора та час їх виконання. Кожен коміт має унікальний ідентифікатор, що дозволяє однозначно визначити стан проєкту. Важливою практикою є створення невеликих, логічно завершених комітів із чіткими повідомленнями, що описують суть змін.

Гілки є механізмом організації паралельної роботи над різними частинами проєкту. Гілка представляє собою окрему лінію розвитку проєкту, що дозволяє виконувати зміни незалежно від основної версії. Використання гілок дозволяє розробникам працювати над новими функціями, виправленнями помилок або експериментальними змінами без впливу на стабільну версію системи.

Основною гілкою зазвичай є гілка, що містить стабільну версію програмного продукту. Додаткові гілки створюються для реалізації окремих задач. Після завершення роботи зміни з гілки можуть бути інтегровані у основну гілку.

Злиття гілок є процесом об'єднання змін з різних гілок у єдину версію проєкту. У Git для цього використовується механізм merge або rebase. Злиття дозволяє інтегрувати результати роботи різних розробників та забезпечити узгодженість коду.

У процесі злиття можуть виникати конфлікти, якщо однакові частини коду були змінені у різних гілках. Конфлікт означає, що система не може автоматично визначити, які зміни є правильними. У такому випадку необхідно вручну виконати розв'язання конфлікту, обравши або об'єднавши відповідні зміни.

Розв'язання конфліктів є важливим етапом інтеграції, що вимагає уважного аналізу змін та розуміння логіки роботи системи. Після вирішення конфлікту необхідно перевірити коректність роботи програми та зафіксувати зміни у новому коміті.

Для організації колективної роботи широко використовуються віддалені репозиторії, розміщені на спеціалізованих платформах, таких як GitHub або GitLab. Вони забезпечують

централізоване зберігання коду, управління доступом та додаткові інструменти для координації роботи команди.

Одним із ключових механізмів узгодження змін є pull request. Це запит на інтеграцію змін з однієї гілки до іншої, зазвичай до основної гілки проєкту. Pull request містить опис змін, що були виконані, а також дозволяє іншим учасникам команди переглянути ці зміни перед їх інтеграцією.

Процес код-рев'ю є важливим елементом забезпечення якості програмного продукту. Він передбачає перевірку коду іншими розробниками з метою виявлення помилок, покращення структури та забезпечення відповідності стандартам проєкту. Код-рев'ю дозволяє підвищити якість програмного коду, забезпечити обмін знаннями між членами команди та запобігти виникненню дефектів.

У процесі код-рев'ю перевіряється коректність реалізації, відповідність вимогам, читабельність коду та дотримання архітектурних принципів. За результатами перевірки можуть бути запропоновані зміни, які необхідно внести перед злиттям гілки.

Організація роботи з використанням систем контролю версій також передбачає визначення правил роботи з репозиторієм. До таких правил належать стандарти іменування гілок, структура комітів, порядок інтеграції змін та вимоги до код-рев'ю. Дотримання єдиних правил забезпечує узгодженість роботи команди та спрощує управління проєктом.

Важливим аспектом є також автоматизація процесів, зокрема використання систем безперервної інтеграції, що дозволяють автоматично перевіряти зміни, виконувати тестування та забезпечувати стабільність системи. Це дозволяє виявляти помилки на ранніх етапах та зменшити ризики під час інтеграції змін.

Таким чином, системи контролю версій, зокрема Git, є ключовим інструментом організації процесу розробки програмних продуктів. Використання репозиторіїв, комітів, гілок та механізмів злиття дозволяє ефективно управляти змінами, забезпечувати колективну роботу та підтримувати якість програмного коду. Застосування pull request і код-рев'ю сприяє узгодженню змін та підвищенню надійності програмного продукту.

Лекція 8. Інструментальні засоби розробки та автоматизація процесу створення програмного продукту

Сучасна розробка програмного забезпечення неможлива без використання інструментальних засобів, що забезпечують ефективну організацію процесу створення програмного продукту. До таких засобів належать середовища програмування, системи збирання, інструменти керування залежностями, засоби автоматизації, безперервної інтеграції та контролю якості коду. Їх використання дозволяє підвищити продуктивність розробників, забезпечити узгодженість процесів та покращити якість програмного продукту.

Середовища програмування є основним інструментом розробника, що забезпечує створення, редагування, налагодження та тестування програмного коду. Сучасні інтегровані середовища розробки поєднують у собі редактор коду, компілятор або інтерпретатор, засоби налагодження та інструменти аналізу. Вони дозволяють автоматизувати рутинні операції, такі як форматування коду, підказки синтаксису та виявлення помилок.

До поширених середовищ програмування належать Visual Studio Code, Visual Studio та CLion. Вибір середовища залежить від мови програмування, типу проєкту та особистих

вподобань розробника. Використання сучасних середовищ дозволяє значно прискорити процес розробки та зменшити кількість помилок.

Системи збирання програмного забезпечення призначені для автоматизації процесу перетворення вихідного коду у виконуваний продукт. Вони забезпечують компіляцію, лінкування, генерацію виконуваних файлів, а також виконання додаткових операцій, таких як тестування або упаковка програмного продукту.

Системи збирання визначають порядок виконання операцій та залежності між ними. Це дозволяє автоматично виконувати лише ті дії, які необхідні для оновлення результату після змін у коді. До популярних систем збирання належать CMake, Make та Gradle.

Керування залежностями є важливою складовою процесу розробки, оскільки сучасні програмні продукти використовують значну кількість сторонніх бібліотек і компонентів. Системи керування залежностями дозволяють автоматично підключати необхідні бібліотеки, контролювати їх версії та забезпечувати узгодженість проєкту.

До таких систем належать npm, pip та Maven. Вони дозволяють визначати залежності у спеціальних конфігураційних файлах та автоматично завантажувати необхідні компоненти. Це значно спрощує процес налаштування середовища розробки та забезпечує відтворюваність проєкту.

Автоматизація збирання є важливим елементом сучасної розробки, оскільки дозволяє виконувати всі необхідні операції без участі користувача. Це включає компіляцію, тестування, перевірку якості коду та створення готових до розгортання пакетів. Автоматизація зменшує ймовірність помилок, пов’язаних із ручним виконанням операцій, та забезпечує стабільність процесу.

Безперервна інтеграція є підходом до організації розробки, що передбачає регулярне об’єднання змін у спільному репозиторії та автоматичну перевірку їх коректності. Кожна зміна у коді запускає процес збирання та тестування, що дозволяє швидко виявляти помилки та забезпечувати стабільність системи.

Для реалізації безперервної інтеграції використовуються спеціалізовані платформи, такі як Jenkins, GitHub Actions та GitLab CI/CD. Вони дозволяють налаштовувати автоматичні процеси, що виконуються при зміні коду, та контролювати їх результати.

Процес безперервної інтеграції включає декілька етапів. Спочатку виконується отримання актуальної версії коду з репозиторію. Далі здійснюється збирання проєкту, виконання тестів та перевірка якості коду. У разі успішного завершення процесу зміни можуть бути інтегровані у основну гілку. У випадку помилок розробник отримує повідомлення, що дозволяє швидко їх виправити.

Контроль якості коду є важливою складовою процесу розробки програмного продукту. Він включає перевірку відповідності коду встановленим стандартам, аналіз його структури та виявлення потенційних помилок. Для цього використовуються інструменти статичного та динамічного аналізу.

Статичний аналіз виконується без запуску програми та дозволяє виявляти синтаксичні помилки, порушення стилю коду та потенційні дефекти. Динамічний аналіз передбачає виконання програми та аналіз її поведінки у процесі роботи. Використання таких підходів дозволяє підвищити надійність програмного продукту.

До інструментів контролю якості належать SonarQube та ESLint. Вони дозволяють автоматично перевіряти код, формувати звіти про його якість та виявляти проблемні місця. Це сприяє підтримці високого рівня якості програмного забезпечення.

Важливим аспектом є інтеграція інструментів контролю якості у процес безперервної інтеграції. Це дозволяє автоматично перевіряти кожну зміну у коді та забезпечувати відповідність стандартам ще на етапі розробки. Такий підхід дозволяє зменшити кількість помилок та підвищити стабільність системи.

Таким чином, інструментальні засоби розробки відіграють ключову роль у створенні програмних продуктів. Використання сучасних середовищ програмування, систем збирання та керування залежностями, а також засобів автоматизації, безперервної інтеграції та контролю якості коду дозволяє підвищити ефективність розробки, забезпечити узгодженість процесів та створювати програмні продукти високої якості.

Лекція 9. Робота з даними у програмних продуктах та організація їх обробки

Робота з даними є центральним аспектом створення більшості сучасних програмних продуктів, оскільки саме дані визначають зміст функціональності системи, її поведінку та цінність для користувача. Ефективна організація зберігання, обробки та передачі даних безпосередньо впливає на продуктивність, надійність та масштабованість програмного забезпечення.

Інформаційна модель є абстрактним представленням даних предметної області, що визначає їх структуру, зв’язки та обмеження. Вона дозволяє формалізувати уявлення про дані, що обробляються системою, та забезпечити узгодженість між різними компонентами програмного продукту. Інформаційна модель визначає, які сутності існують у системі, які атрибути вони мають та як пов’язані між собою.

Побудова інформаційної моделі є важливим етапом проєктування, оскільки вона визначає основу для реалізації структури даних у програмному коді та базі даних. Вона може бути представлена у вигляді діаграм, таблиць або формалізованих описів, що дозволяють відобразити логічну структуру даних.

Організація зберігання даних передбачає вибір способів та засобів їх збереження у програмній системі. Залежно від типу програмного продукту можуть використовуватися різні підходи, включаючи файлові системи, реляційні бази даних, нереляційні сховища та розподілені системи зберігання.

Реляційні бази даних базуються на табличній моделі представлення даних, де інформація зберігається у вигляді таблиць, що складаються з рядків та стовпців. Вони забезпечують цілісність даних, підтримку транзакцій та можливість виконання складних запитів. Такий підхід є доцільним для систем, де важлива структурованість та узгодженість даних.

Нереляційні бази даних використовуються у випадках, коли необхідна висока масштабованість, гнучкість структури даних або обробка великих обсягів інформації. Вони можуть використовувати різні моделі даних, зокрема документну, ключ-значення або графову. Вибір типу сховища залежить від вимог до системи та характеру даних.

Обробка даних включає виконання операцій над даними, таких як введення, збереження, зміна, видалення та аналіз. Вона реалізується у програмному коді та забезпечує виконання функціональних вимог системи. Важливою є ефективність обробки, що визначає швидкість виконання операцій та продуктивність системи.

Валідація даних є процесом перевірки їх коректності перед використанням або збереженням. Вона дозволяє забезпечити відповідність даних встановленим вимогам та

запобігти виникненню помилок у роботі системи. Валідація може включати перевірку типів даних, діапазонів значень, форматів та логічних обмежень.

Валідація може виконуватися як на рівні інтерфейсу користувача, так і на рівні серверної частини системи. Перевірка на клієнтському рівні дозволяє швидко виявляти помилки та покращує зручність використання системи, тоді як серверна валідація забезпечує надійність та безпеку обробки даних.

Взаємодія з базами даних є важливою складовою програмних продуктів, що потребують зберігання та обробки інформації. Вона реалізується за допомогою спеціалізованих мов запитів або програмних інтерфейсів. У випадку реляційних баз даних широко використовується мова SQL, що дозволяє виконувати операції над даними та керувати структурою бази.

Для спрощення взаємодії з базами даних використовуються бібліотеки та фреймворки, що реалізують механізми відображення об’єктів програмного коду на структури бази даних. Це дозволяє працювати з даними у вигляді об’єктів, що спрощує розробку та підвищує зрозумілість коду.

Взаємодія із зовнішніми сервісами є важливою складовою сучасних програмних систем, оскільки багато функцій реалізується через використання сторонніх ресурсів. Це можуть бути вебсервіси, платіжні системи, системи авторизації або інші сервіси.

Взаємодія із зовнішніми сервісами здійснюється через мережеві інтерфейси, що визначають правила обміну даними. Найбільш поширеним є використання протоколу HTTP та форматів даних, таких як JSON або XML. Це дозволяє забезпечити сумісність між різними системами та спростити інтеграцію.

Під час взаємодії із зовнішніми сервісами важливо враховувати питання безпеки, зокрема автентифікацію, авторизацію та захист переданих даних. Використання захищених протоколів та механізмів контролю доступу дозволяє забезпечити конфіденційність та цілісність інформації.

Також необхідно враховувати надійність взаємодії, оскільки зовнішні сервіси можуть бути недоступними або працювати з помилками. Для цього використовуються механізми повторних спроб, обробки помилок та резервування. Це дозволяє забезпечити стабільність роботи системи навіть у разі збоїв зовнішніх компонентів.

Організація роботи з даними повинна враховувати вимоги до продуктивності, безпеки та масштабованості. Використання кешування дозволяє зменшити навантаження на базу даних та прискорити доступ до часто використовуваної інформації. Оптимізація запитів та індексація даних забезпечують ефективність виконання операцій.

Таким чином, робота з даними є ключовим аспектом створення програмних продуктів, що визначає їх функціональність та ефективність. Використання інформаційних моделей, правильна організація зберігання даних, забезпечення їх обробки та валідації, а також ефективна взаємодія з базами даних та зовнішніми сервісами дозволяють створювати надійні та продуктивні програмні системи.

ТЕМА 4. ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Лекція 10. Якість програмного забезпечення та стандарти у програмній інженерії

Якість програмного забезпечення є однією з ключових характеристик програмного продукту, що визначає його придатність до використання відповідно до встановлених вимог. У сучасній інженерії програмного забезпечення якість розглядається як сукупність властивостей системи, що забезпечують її відповідність функціональним і нефункціональним вимогам, а також очікуванням користувачів.

Поняття якості програмного забезпечення охоплює не лише коректність виконання функцій, але й такі характеристики, як надійність, продуктивність, зручність використання та можливість супроводу. Якість є багатовимірним поняттям, що формується на всіх етапах життєвого циклу програмного продукту, починаючи від формування вимог і закінчуючи його експлуатацією.

Атрибути якості визначають конкретні характеристики програмного забезпечення, що підлягають оцінюванню. Вони дозволяють формалізувати поняття якості та забезпечити можливість її вимірювання. До основних атрибутів якості належать функціональна придатність, надійність, продуктивність, зручність використання, супроводжуваність та переносимість.

Функціональна придатність визначає ступінь відповідності програмного продукту встановленим функціональним вимогам. Вона характеризує, наскільки система виконує необхідні функції та забезпечує досягнення поставлених цілей.

Надійність визначає здатність системи стабільно функціонувати протягом певного часу без збоїв. Вона включає такі характеристики, як відмовостійкість, здатність до відновлення та стабільність роботи.

Продуктивність характеризує ефективність використання ресурсів системи, зокрема швидкість обробки даних та час відповіді. Вона є важливою для систем, що працюють у реальному часі або обробляють великі обсяги інформації.

Зручність використання визначає легкість взаємодії користувача з системою. Вона включає зрозумілість інтерфейсу, простоту навчання та ефективність виконання задач.

Супроводжуваність визначає можливість внесення змін у програмний продукт, включаючи виправлення помилок, додавання нової функціональності та адаптацію до змін у середовищі. Вона залежить від структури коду, документації та рівня модульності системи.

Переносимість визначає здатність програмного забезпечення працювати у різних середовищах без значних змін. Вона є важливою для систем, що повинні підтримувати різні платформи або конфігурації.

Моделі якості використовуються для систематизації атрибутів якості та визначення їх взаємозв'язків. Вони дозволяють формалізувати підхід до оцінювання якості та забезпечити єдине розуміння цього поняття.

Однією з найбільш відомих моделей якості є ISO/IEC 25010, що визначає набір характеристик якості програмного забезпечення та їх підхарактеристики. Ця модель використовується для оцінювання якості як готового продукту, так і процесів його розробки.

Модель ISO/IEC 25010 включає такі основні характеристики, як функціональна придатність, надійність, продуктивність, сумісність, зручність використання, безпека,

супроводжуваність та переносимість. Вона є універсальним інструментом для оцінювання якості програмного забезпечення та широко використовується у практиці розробки.

Стандарти у сфері програмної інженерії визначають правила, рекомендації та вимоги до процесів розробки, якості та управління програмним забезпеченням. Вони забезпечують узгодженість процесів, підвищують рівень якості та сприяють ефективній взаємодії між учасниками проєкту.

До важливих стандартів належать ISO/IEC 12207, що визначає процеси життєвого циклу програмного забезпечення, та ISO/IEC 15504, що використовується для оцінювання зрілості процесів розробки. Використання таких стандартів дозволяє забезпечити системний підхід до управління якістю програмного продукту.

Вимоги до якості формуються на різних етапах життєвого циклу програмного забезпечення та змінюються залежно від етапу розробки. На етапі формування вимог визначаються критерії якості, що мають бути досягнуті у програмному продукті. Вони включають як функціональні, так і нефункціональні вимоги.

На етапі проєктування визначаються архітектурні рішення, що забезпечують досягнення необхідних характеристик якості. Важливо враховувати вимоги до масштабованості, надійності та безпеки при виборі архітектури системи.

На етапі реалізації здійснюється контроль якості програмного коду, включаючи дотримання стандартів програмування, використання ефективних алгоритмів та забезпечення читабельності коду. Це впливає на супроводжуваність та надійність системи.

На етапі тестування перевіряється відповідність програмного продукту встановленим вимогам до якості. Використовуються різні методи тестування, що дозволяють виявити дефекти та оцінити характеристики системи.

Під час впровадження та експлуатації важливим є забезпечення стабільності роботи системи, моніторинг її стану та своєчасне реагування на проблеми. На цьому етапі також здійснюється збір зворотного зв’язку від користувачів, що дозволяє оцінити якість програмного продукту в реальних умовах.

Супровід програмного забезпечення передбачає підтримку його якості протягом усього періоду експлуатації. Це включає виправлення помилок, оновлення та адаптацію до змін у середовищі. Важливим є забезпечення зворотної сумісності та стабільності системи.

Таким чином, якість програмного забезпечення є комплексною характеристикою, що формується на всіх етапах його створення та експлуатації. Використання моделей якості, дотримання стандартів програмної інженерії та забезпечення відповідності вимогам до якості дозволяють створювати надійні, ефективні та конкурентоспроможні програмні продукти.

Лекція 11. Тестування програмного забезпечення та організація перевірки якості

Тестування програмного забезпечення є одним із ключових процесів у створенні програмного продукту, що спрямований на виявлення дефектів, перевірку відповідності вимогам та забезпечення належного рівня якості системи. Воно є невід’ємною складовою життєвого циклу програмного забезпечення та виконується на різних етапах розробки.

Під тестуванням розуміють процес виконання програмного продукту з метою виявлення помилок та перевірки його поведінки у різних умовах. Важливо розуміти, що тестування не гарантує повної відсутності дефектів, але дозволяє зменшити їх кількість та оцінити рівень якості системи.

Тестування виконує декілька функцій, серед яких перевірка правильності реалізації функціональності, виявлення дефектів, оцінювання характеристик якості та забезпечення довіри до програмного продукту. Воно також дозволяє перевірити відповідність системи вимогам та забезпечити її готовність до експлуатації.

Рівні тестування визначають, на яких етапах і в якому обсязі виконується перевірка програмного забезпечення. Вони відображають різні аспекти системи та дозволяють поетапно перевіряти її компоненти.

Модульне тестування є найнижчим рівнем тестування та спрямоване на перевірку окремих компонентів або модулів програмного забезпечення. Воно дозволяє виявити помилки на ранніх етапах розробки та забезпечити коректність роботи окремих частин системи. Модульні тести зазвичай виконуються розробниками та можуть бути автоматизованими.

Інтеграційне тестування спрямоване на перевірку взаємодії між окремими компонентами системи. Воно дозволяє виявити помилки, пов’язані з обміном даними, узгодженістю інтерфейсів та інтеграцією модулів. Інтеграційне тестування може виконуватися поступово, у міру об’єднання компонентів у систему.

Системне тестування передбачає перевірку всієї системи як єдиного цілого. Воно спрямоване на оцінювання відповідності програмного продукту встановленим вимогам, включаючи функціональні та нефункціональні характеристики. На цьому рівні перевіряється поведінка системи у реальних умовах експлуатації.

Приймальне тестування є завершальним етапом перевірки програмного продукту перед його впровадженням. Воно виконується з метою підтвердження відповідності системи вимогам замовника. Приймальне тестування може виконуватися як розробниками, так і представниками замовника або кінцевими користувачами.

Тестові сценарії є формалізованим описом послідовності дій, що виконуються під час тестування. Вони визначають умови виконання тесту, вхідні дані, очікувані результати та критерії оцінювання. Тестові сценарії дозволяють забезпечити системність та повторюваність процесу тестування.

На основі тестових сценаріїв формуються тестові випадки, що містять конкретні значення вхідних даних та очікувані результати. Вони використовуються для перевірки різних аспектів функціональності системи та дозволяють виявляти дефекти.

Важливим аспектом є повнота покриття тестування, що визначає, наскільки повно перевірено функціональність системи. Для цього використовуються різні критерії, зокрема покриття коду, покриття умов та покриття сценаріїв. Забезпечення достатнього рівня покриття дозволяє підвищити якість програмного продукту.

Автоматизоване тестування є підходом, що передбачає використання програмних засобів для виконання тестів без участі людини. Воно дозволяє значно прискорити процес тестування, підвищити його точність та забезпечити можливість регулярного виконання тестів.

Автоматизовані тести можуть використовуватися на різних рівнях тестування, зокрема для модульного та інтеграційного тестування. Вони дозволяють швидко перевіряти зміни у коді та виявляти дефекти на ранніх етапах розробки. Використання автоматизованого тестування є особливо важливим у процесах безперервної інтеграції.

Разом з тим, не всі види тестування можуть бути повністю автоматизованими. Наприклад, тестування зручності використання або оцінювання інтерфейсу користувача

часто потребує участі людини. Тому ефективний процес тестування передбачає поєднання автоматизованих та ручних методів.

Організація процесу тестування передбачає планування, визначення обсягу робіт, розподіл ресурсів та контроль виконання. Важливим є визначення стратегії тестування, що враховує особливості програмного продукту та вимоги до якості.

У процесі тестування також здійснюється документування результатів, що дозволяє відстежувати виявлені дефекти, оцінювати їх критичність та контролювати процес їх усунення. Це забезпечує прозорість процесу та дозволяє приймати обґрунтовані рішення щодо готовності системи до експлуатації.

Таким чином, тестування програмного забезпечення є необхідним етапом створення програмного продукту, що забезпечує перевірку його якості та відповідності вимогам. Використання різних рівнів тестування, формування тестових сценаріїв та застосування автоматизованих засобів дозволяє підвищити надійність системи та забезпечити її ефективну роботу у реальних умовах експлуатації.

Лекція 12. Верифікація та валідація програмного забезпечення. Аналіз коду та управління дефектами

Верифікація та валідація є взаємопов’язаними процесами забезпечення якості програмного забезпечення, що виконуються протягом усього життєвого циклу програмного продукту. Вони дозволяють оцінити відповідність системи встановленим вимогам та очікуванням користувачів, а також забезпечити правильність її реалізації.

Верифікація спрямована на перевірку того, чи правильно реалізовано програмний продукт відповідно до заданих специфікацій. Вона відповідає на питання, чи відповідає система визначеним вимогам. Верифікація виконується на різних етапах розробки та включає аналіз документації, перевірку проєктних рішень, тестування та аналіз програмного коду.

Валідація, у свою чергу, спрямована на перевірку того, чи відповідає програмний продукт потребам користувачів та замовника. Вона відповідає на питання, чи створено правильний продукт. Валідація зазвичай виконується на завершальних етапах розробки та включає приймальне тестування, оцінювання функціональності та аналіз поведінки системи у реальних умовах.

Верифікація та валідація доповнюють одна одну та забезпечують комплексний підхід до контролю якості програмного забезпечення. Їх ефективне поєднання дозволяє зменшити кількість дефектів та забезпечити відповідність програмного продукту вимогам.

Статичний аналіз коду є методом перевірки програмного забезпечення без його виконання. Він дозволяє виявляти синтаксичні помилки, порушення стандартів програмування, потенційні дефекти та проблеми у структурі коду. Статичний аналіз виконується за допомогою спеціалізованих інструментів, що автоматично перевіряють код.

Статичний аналіз дозволяє виявляти помилки на ранніх етапах розробки, що зменшує витрати на їх виправлення. Він також сприяє підвищенню якості коду, забезпечуючи дотримання стандартів та рекомендацій. До типових проблем, що виявляються статичним аналізом, належать некоректні конструкції, потенційні витоки пам’яті, невикористані змінні та інші дефекти.

Динамічний аналіз передбачає перевірку програмного забезпечення під час його виконання. Він дозволяє оцінити поведінку системи у реальних умовах, виявити помилки виконання, проблеми з продуктивністю та інші дефекти, що не можуть бути виявлені статичним аналізом.

Динамічний аналіз включає виконання тестів, моніторинг роботи системи, аналіз використання ресурсів та виявлення помилок під час виконання. Він є важливим для оцінювання реальної поведінки програмного продукту та забезпечення його надійності.

Поєднання статичного та динамічного аналізу дозволяє забезпечити більш повне покриття процесу перевірки програмного забезпечення. Статичний аналіз дозволяє виявити потенційні проблеми на ранніх етапах, тоді як динамічний аналіз дозволяє перевірити фактичну поведінку системи.

Документування результатів тестування є важливою складовою процесу забезпечення якості. Воно дозволяє фіксувати результати перевірок, виявлені дефекти та їх статус. Документація забезпечує прозорість процесу тестування та дозволяє відстежувати прогрес у виправленні помилок.

Документація може включати звіти про тестування, журнали дефектів, результати виконання тестів та іншу інформацію, що використовується для аналізу якості програмного продукту. Вона також є основою для прийняття рішень щодо готовності системи до впровадження.

Усунення дефектів є процесом виправлення помилок, виявлених під час тестування та аналізу. Він включає визначення причин виникнення дефекту, внесення змін у програмний код та перевірку правильності виправлення. Важливою є систематизація дефектів, що дозволяє визначити їх критичність та пріоритет виправлення.

Контроль якості передбачає постійний моніторинг процесу розробки та результатів тестування з метою забезпечення відповідності програмного продукту встановленим вимогам. Він включає аналіз процесів, перевірку відповідності стандартам та оцінювання ефективності розробки.

Ефективний контроль якості передбачає використання показників, що дозволяють оцінити стан програмного продукту, зокрема кількість дефектів, їх критичність, час виправлення та рівень покриття тестування. Це дозволяє своєчасно виявляти проблеми та приймати обґрунтовані рішення.

Важливим аспектом є інтеграція процесів верифікації, валідації та контролю якості у загальний процес розробки. Це дозволяє забезпечити безперервний контроль якості та своєчасне виявлення дефектів.

Таким чином, верифікація та валідація, а також використання методів статичного та динамічного аналізу, документування результатів тестування та ефективне управління дефектами є необхідними складовими процесу створення програмного продукту. Вони забезпечують високий рівень якості програмного забезпечення та його відповідність вимогам і очікуванням користувачів.

Тема 5. Управління проєктами та супроводження програмних продуктів

Лекція 13. Основи управління ІТ-проєктами та організація процесу розробки програмного продукту

Управління ІТ-проєктами є важливою складовою процесу створення програмних продуктів, оскільки забезпечує організацію роботи команди, ефективне використання ресурсів та досягнення поставлених цілей у визначені терміни. На відміну від суто технічної діяльності розробки, управління проєктом охоплює планування, координацію, контроль та аналіз виконання робіт.

ІТ-проєкт характеризується наявністю визначеної мети, обмежених ресурсів, часових рамок та певного рівня невизначеності. Особливістю таких проєктів є висока складність, динамічність вимог та залежність від людського фактора. Тому ефективне управління проєктом передбачає системний підхід до організації процесів та прийняття рішень.

Планування робіт є початковим етапом управління проєктом, що визначає структуру майбутньої діяльності. Воно включає визначення обсягу робіт, їх розподіл на окремі задачі, встановлення послідовності виконання та оцінювання необхідних ресурсів. Важливим є формування чіткого уявлення про кінцевий результат та проміжні етапи його досягнення.

Одним із підходів до планування є декомпозиція проєкту на підзадачі. Це дозволяє зменшити складність управління та забезпечити контроль виконання окремих елементів. Кожна задача повинна мати визначений результат, терміни виконання та відповідального виконавця. Такий підхід дозволяє підвищити прозорість процесу та спростити контроль виконання.

Під час планування також визначаються залежності між задачами, що впливають на порядок їх виконання. Наявність залежностей вимагає координації робіт та врахування обмежень, пов’язаних із ресурсами та часовими рамками. Це дозволяє сформувати реалістичний графік виконання проєкту.

Оцінювання трудомісткості є важливим елементом планування, що дозволяє визначити обсяг ресурсів, необхідних для виконання проєкту. Трудомісткість визначається на основі аналізу складності задач, досвіду команди та використаних технологій. Вона може оцінюватися у людино-годинах або інших одиницях вимірювання.

Існують різні підходи до оцінювання трудомісткості, зокрема експертні оцінки, аналогія з попередніми проєктами та використання формалізованих моделей. Важливо враховувати невизначеність та можливі відхилення, що можуть виникати у процесі виконання робіт. Тому оцінки повинні містити резерв часу та ресурсів.

У гнучких методологіях часто використовується відносне оцінювання, що базується на порівнянні складності задач. Це дозволяє швидше виконувати оцінювання та враховувати особливості конкретного проєкту. Незалежно від обраного підходу, точність оцінювання має суттєвий вплив на успішність проєкту.

Управління ризиками є процесом виявлення, аналізу та мінімізації факторів, що можуть негативно вплинути на виконання проєкту. Ризики можуть бути пов’язані з технічними аспектами, організаційними проблемами, змінами вимог або зовнішніми факторами.

Процес управління ризиками включає ідентифікацію ризиків, оцінювання їх ймовірності та впливу, а також розробку заходів щодо їх зниження. Важливо не лише

реагувати на вже виниклі проблеми, але й передбачати потенційні загрози. Це дозволяє підвищити стійкість проєкту та зменшити негативні наслідки.

Для кожного ризику визначаються стратегії реагування, які можуть включати уникнення, зменшення, передавання або прийняття ризику. Наприклад, технічні ризики можуть бути зменшені шляхом проведення прототипування, а організаційні – за рахунок покращення комунікації у команді.

Контроль виконання завдань є постійним процесом, що дозволяє відстежувати прогрес проєкту та своєчасно виявляти відхилення від плану. Він включає аналіз виконаних робіт, оцінювання продуктивності команди та порівняння фактичних результатів із запланованими.

Для ефективного контролю використовуються показники, що відображають стан проєкту, зокрема відсоток виконання задач, використання ресурсів та відповідність термінам. Регулярний контроль дозволяє оперативно вносити корективи у план робіт та забезпечувати досягнення поставлених цілей.

Важливою складовою є організація зворотного зв'язку, що дозволяє враховувати результати виконання робіт та покращувати процес управління. Це може включати проведення регулярних нарад, аналіз виконаних задач та обговорення проблем, що виникають у процесі розробки.

Інструменти управління проєктами забезпечують автоматизацію процесів планування, контролю та координації робіт. Вони дозволяють створювати списки задач, визначати їх пріоритети, відстежувати виконання та забезпечувати взаємодію між членами команди.

До поширених інструментів належать Jira, Trello та Asana. Вони надають можливість візуалізації процесу роботи, управління завданнями та контролю виконання проєкту.

Використання таких інструментів дозволяє забезпечити прозорість процесу, підвищити ефективність взаємодії між учасниками проєкту та зменшити ймовірність помилок. Вони також дозволяють зберігати історію виконання задач та аналізувати результати роботи.

У сучасних умовах управління IT-проєктами часто поєднує елементи різних підходів, зокрема традиційних та гнучких методологій. Це дозволяє адаптувати процес управління до конкретних умов проєкту та забезпечити оптимальне використання ресурсів.

Таким чином, управління IT-проєктами є складним та багатогранним процесом, що включає планування, оцінювання трудомісткості, управління ризиками, контроль виконання завдань та використання спеціалізованих інструментів. Ефективна організація цих процесів дозволяє забезпечити успішну реалізацію програмного продукту, відповідність вимогам та досягнення поставлених цілей.

Лекція 14. Управління змінами та конфігурацією програмного забезпечення

У процесі створення та експлуатації програмного продукту зміни є неминучими, оскільки змінюються вимоги, середовище функціонування, технології та потреби користувачів. Управління змінами та конфігурацією програмного забезпечення є ключовими процесами, що забезпечують контрольоване внесення змін, збереження цілісності системи та підтримку її стабільної роботи.

Управління змінами передбачає організацію процесу внесення змін у програмний продукт, включаючи їх ініціювання, аналіз, затвердження, реалізацію та контроль

результатів. Основною метою є забезпечення того, щоб усі зміни були обґрунтованими, узгодженими та не призводили до порушення функціональності системи.

Зміни можуть виникати з різних причин, зокрема у зв'язку з уточненням вимог, виявленням дефектів, необхідністю підвищення продуктивності або адаптації до нових умов експлуатації. Кожна зміна повинна бути формалізована у вигляді запиту, що містить опис проблеми, обґрунтування необхідності зміни та очікувані результати.

Процес управління змінами включає декілька етапів. На першому етапі здійснюється реєстрація запиту на зміну. Далі виконується аналіз зміни, що передбачає оцінювання її впливу на систему, визначення необхідних ресурсів та ризиків. На основі цього приймається рішення щодо доцільності впровадження зміни.

Після затвердження зміни виконується її реалізація, що включає внесення змін у програмний код, тестування та інтеграцію у систему. Важливим є контроль результатів, що дозволяє переконатися у правильності реалізації та відсутності негативного впливу на інші компоненти системи.

Управління конфігурацією програмного забезпечення є процесом визначення, організації та контролю компонентів системи та їх версій. Конфігурація включає всі артефакти, що входять до складу програмного продукту, зокрема програмний код, документацію, бібліотеки, налаштування та інші елементи.

Основною метою управління конфігурацією є забезпечення узгодженості між різними компонентами системи та можливість відтворення будь-якої версії програмного продукту. Це досягається за рахунок використання систем контролю версій, чіткої ідентифікації компонентів та ведення історії змін.

Важливим аспектом є визначення конфігураційних одиниць, що представляють окремі елементи системи, які підлягають контролю. Для кожної конфігураційної одиниці визначається її версія, стан та взаємозв'язки з іншими компонентами. Це дозволяє ефективно управляти складною структурою програмного продукту.

Реліз-менеджмент є процесом планування, підготовки та випуску нових версій програмного продукту. Він передбачає координацію всіх етапів, пов'язаних із створенням релізу, включаючи інтеграцію змін, тестування, підготовку документації та розгортання.

Процес реліз-менеджменту включає визначення складу релізу, що містить перелік змін, які будуть включені у нову версію. Далі виконується підготовка релізу, що включає інтеграцію змін, проведення тестування та усунення виявлених дефектів. Після цього виконується розгортання релізу у робочому середовищі.

Важливим є також контроль стабільності релізу після його впровадження. Це передбачає моніторинг роботи системи, аналіз можливих проблем та своєчасне реагування на них. У разі необхідності можуть виконуватися виправлення або відкат до попередньої версії.

Підтримка цілісності та стабільності системи є одним із основних завдань управління змінами та конфігурацією. Цілісність означає узгодженість усіх компонентів системи та відсутність суперечностей між ними. Стабільність визначає здатність системи працювати без збоїв у процесі експлуатації.

Для забезпечення цілісності використовуються механізми контролю залежностей, перевірки сумісності компонентів та автоматизованого тестування. Це дозволяє виявляти проблеми на ранніх етапах та запобігати їх впливу на систему.

Стабільність системи забезпечується за рахунок використання перевірених версій компонентів, проведення комплексного тестування та впровадження змін у контрольованому

середовищі. Важливим є також використання резервних копій та механізмів відновлення, що дозволяють швидко реагувати на збої.

Документування змін є необхідною складовою процесу управління конфігурацією. Воно передбачає фіксацію всіх змін, що були внесені у систему, включаючи їх опис, причини та результати. Це дозволяє відстежувати історію розвитку програмного продукту та забезпечує прозорість процесу.

Документація змін може включати журнали змін, опис релізів, звіти про виправлення дефектів та інші матеріали. Вона використовується як розробниками, так і користувачами для розуміння функціональних можливостей системи та особливостей її використання.

Важливим є забезпечення актуальності документації, що передбачає своєчасне оновлення інформації при внесенні змін. Це дозволяє уникнути розбіжностей між документацією та фактичним станом системи.

Таким чином, управління змінами та конфігурацією є критично важливими процесами, що забезпечують контрольоване внесення змін, підтримку цілісності та стабільності програмного продукту, а також ефективну організацію його розвитку. Використання структурованих підходів до управління змінами, реліз-менеджменту та документування дозволяє забезпечити високу якість програмного забезпечення та його відповідність вимогам користувачів.

Лекція 15. Супроводження програмного забезпечення та підтримка програмних продуктів

Супроводження програмного забезпечення є завершальним, але водночас одним із найдовших етапів життєвого циклу програмного продукту. Після впровадження система переходить у стадію експлуатації, під час якої виникає необхідність її підтримки, удосконалення та адаптації до нових умов. У практиці інженерії програмного забезпечення витрати на супроводження можуть перевищувати витрати на початкову розробку, що підкреслює важливість правильного організаційного підходу до цього процесу.

Супроводження програмного забезпечення включає комплекс дій, спрямованих на забезпечення працездатності системи, усунення дефектів, удосконалення функціональності та підтримку відповідності вимогам користувачів. Воно виконується після впровадження системи та може тривати протягом усього періоду її використання.

Коригувальне супроводження передбачає виправлення дефектів, виявлених під час експлуатації програмного продукту. Дефекти можуть виникати внаслідок помилок у програмному коді, неточностей у вимогах або змін у середовищі виконання. Основною метою коригувального супроводження є відновлення коректної роботи системи та забезпечення її стабільності.

Адаптивне супроводження спрямоване на пристосування програмного продукту до змін у зовнішньому середовищі. Це можуть бути зміни у апаратному забезпеченні, операційних системах, базах даних або інтегрованих сервісах. У сучасних умовах, коли технології швидко розвиваються, адаптивне супроводження є необхідною умовою підтримки актуальності програмного продукту.

Еволюційне супроводження передбачає розширення функціональності системи відповідно до нових вимог користувачів або бізнес-потреб. Воно включає додавання нових

можливостей, оптимізацію існуючих функцій та розвиток програмного продукту. Еволюційне супроводження є важливим для підтримки конкурентоспроможності системи.

Супроводження програмного забезпечення тісно пов’язане з масштабуванням систем. Масштабування передбачає забезпечення можливості ефективної роботи системи при зростанні навантаження або обсягу даних. У процесі експлуатації програмного продукту часто виникає необхідність збільшення кількості користувачів, обсягу оброблюваних даних або функціональності системи.

Масштабування може виконуватися за рахунок оптимізації програмного коду, зміни архітектури або використання додаткових ресурсів. Важливим є забезпечення того, щоб система могла розвиватися без значного зниження продуктивності або стабільності.

Технічна підтримка є складовою супроводження програмного забезпечення та передбачає надання допомоги користувачам у процесі використання системи. Вона включає обробку запитів користувачів, усунення проблем, консультації та навчання.

Організація технічної підтримки може передбачати різні рівні обслуговування, що відрізняються складністю задач та рівнем кваліфікації фахівців. Наприклад, перший рівень може включати обробку типових запитів, тоді як більш складні проблеми передаються на наступні рівні підтримки.

Професійна документація є необхідною складовою супроводження програмного забезпечення. Вона включає технічну документацію, інструкції для користувачів, опис інтерфейсів та інші матеріали, що забезпечують розуміння та ефективне використання системи.

Технічна документація містить інформацію про структуру системи, принципи її роботи та особливості реалізації. Вона використовується розробниками для підтримки та розвитку програмного продукту. Документація для користувачів забезпечує можливість ефективного використання системи без необхідності звернення до розробників.

Важливим є забезпечення актуальності документації, оскільки зміни у програмному продукті повинні відображатися у відповідних документах. Це дозволяє уникнути непорозумінь та забезпечити ефективну взаємодію між користувачами та розробниками.

Взаємодія із замовником є важливою складовою процесу супроводження. Вона включає збір зворотного зв’язку, узгодження змін та визначення напрямів розвитку програмного продукту. Ефективна комунікація дозволяє враховувати потреби користувачів та забезпечувати відповідність системи їх очікуванням.

Зворотний зв’язок може отримуватися через різні канали, зокрема звернення до служби підтримки, опитування або аналіз використання системи. Це дозволяє визначити проблемні місця та можливості для вдосконалення.

У процесі взаємодії із замовником важливо забезпечити прозорість процесів, чіткість формулювання вимог та узгодженість рішень. Це дозволяє уникнути конфліктів та забезпечити ефективну співпрацю.

Таким чином, супроводження програмного забезпечення є невід’ємною частиною життєвого циклу програмного продукту, що забезпечує його стабільну роботу, розвиток та відповідність вимогам користувачів. Використання різних видів супроводження, забезпечення масштабування, організація технічної підтримки, ведення професійної документації та ефективна взаємодія із замовником дозволяють підтримувати високу якість програмного продукту протягом усього періоду його експлуатації.

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

Опорний конспект лекцій для здобувачів вищої освіти,
які навчаються за спеціальностями галузі «Інформаційні технології»

Українською мовою