МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ
з дисципліни
«Методологія інженерії програмного забезпечення»
для студентів спеціальності
121 «Інженерія програмного забезпечення»

Затверджено
редакційно-видавничою
радою університету,
протокол № 1 від 13.02.2025 р.

Харків

НТУ «ХПІ»

2025

Методичні рекомендації з дисципліни «Методологія інженерії програмного забезпечення» для студентів спеціальності 121 «Інженерія програмного забезпечення» / О.Ю. Чередніченко, О.В. Шматко, Ю.С. Літвінова, О.О. Сутягін; М-во освіти і науки України, Нац. техн. ун-т “Харківський політехнічний інститут”. – Харків : НТУ «ХПІ», 2025, - 36 с.

Автори:

О.Ю. Чередніченко, д.т.н., проф.;

О.В. Шматко, к.т.н, доц.;

Ю.С. Літвінова, к. т. н, доц.;

О.О. Сутягін.

Рецензент:

В.В. Москаленко

Кафедра програмної інженерії та інтелектуальних технологій

ЗМІСТ

Вступ 4
1 Структура теоретичного модуля 5
1. Вступ до програмної інженерії 5
2. Аналіз та характеристика областей знань SWEBOK 6
3. Життєві цикли розробки програмного забезпечення (ЖЦ ПЗ) 7
4. Методи визначення вимог у програмній інженерії 8
5. Методи аналізу і побудування моделей програмних об’єктів (ПрО) 10
6. Методи проєктування програмних систем 14
7.Інженерія ПВК (програмних відтворюваних компонентів) 18
8. Методи тестування і верифікації ПЗ 21
9. Інтеграція та змінення компонентів ПЗ 24
10.Реінженерія програмних систем 25
11. Методи управління проєктом, ризиком і конфігурацією 31
Завдання 1. Самостійна робота. 34
Завдання 2. Реферат. 34

ВСТУП

Дисципліна "Методологія інженерії програмного забезпечення" відіграє ключову роль у формуванні висококваліфікованих фахівців, здатних створювати ефективні програмні рішення, які відповідають сучасним стандартам.

Методологія інженерії програмного забезпечення охоплює широкий спектр методів, підходів і практик, спрямованих на оптимізацію всього життєвого циклу програмного продукту – від формулювання вимог і проектування до тестування, розгортання та подальшої підтримки. Вивчення цієї дисципліни дозволяє студентам опанувати не лише теоретичні основи, а й здобути практичні навички використання сучасних технологій, інструментів і методологій, таких як Agile, Scrum, Kanban, DevOps, а також моделей розробки, орієнтованих на якість (TDD, BDD).

Методичні рекомендації, представлені в цьому документі, спрямовані на те, щоб зробити процес навчання максимально структурованим, послідовним і наближеним до реальних потреб галузі. У рекомендаціях детально висвітлюються ключові поняття, концепції та методи, а також надаються приклади їх застосування в реальних проєктах.

Дсципліна є важливою складовою підготовки магістрів, оскільки дозволяє не лише поглибити знання з технічних аспектів розробки програмного забезпечення, а й сформувати системне бачення управління програмними проєктами, враховуючи їхні технічні, економічні, організаційні та соціальні аспекти.

Методичні рекомендації можуть бути використані як базовий інструмент для викладачів, які викладають дисципліну, так і для студентів, що прагнуть опанувати її самостійно. Документ включає як теоретичний матеріал, так і практичні завдання, які допоможуть закріпити знання та застосувати їх на практиці. Зрештою, його мета полягає у сприянні формуванню у студентів необхідних компетенцій для успішної професійної діяльності у сфері розробки програмного забезпечення.

1 СТРУКТУРА ТЕОРЕТИЧНОГО МОДУЛЯ

1. Вступ до програмної інженерії

Програмна інженерія вивчає принципи, методи та підходи до розробки програмного забезпечення, орієнтованого на високу якість і відповідність вимогам користувачів. Вона поєднує науку, технології та інженерний підхід.

Визначення програмної інженерії:

Програмна інженерія включає планування, проєктування, розробку, тестування та підтримку програмних продуктів. Метою програмної інженерії є створення систем, які є ефективними, масштабованими, надійними та простими у використанні. Цей напрямок також акцентує увагу на командній роботі та управлінні проєктами, що важливо для складних систем. Наприклад, розробка банківської системи, що відповідає вимогам безпеки і масштабується для обслуговування мільйонів користувачів є доволі нетривіальною та комплексною задачею для будь-якого бізнесу.

Історія розвитку програмної інженерії:

Програмна інженерія як дисципліна сформувалася у 1960-х роках, коли розробка програмного забезпечення почала стикатися з проблемами масштабованості та складності. Спочатку домінували процедурні та водоспадні моделі, які вимагали повного завершення кожного етапу розробки перед переходом до наступного. У 1980-х роках виникли об'єктно-орієнтовані методи, які дозволили моделювати програмне забезпечення у вигляді об'єктів, що спростило управління складними проєктами. У 2000-х роках з'явилися гнучкі методології, такі як Agile, які дозволили адаптувати розробку до швидко змінюваних вимог. Наприклад, методологія Scrum стала стандартом у створенні веб застосунків завдяки своїй ітеративній природі. Перші програмні системи для космічних апаратів NASA створювалися за водоспадною моделлю.

Основні завдання та виклики:

Програмна інженерія вирішує завдання, які пов'язані з розробкою програмного забезпечення, що задовольняє потреби користувачів і бізнесу. Серед основних завдань — оптимізація ресурсів, забезпечення якості, підтримка безпеки та адаптивності систем. Важливим викликом є керування складністю проєктів, особливо в умовах глобальної розробки, коли команди працюють у різних часових поясах. Іншою

проблемою є швидка втрата акутальності технологій, яка вимагає постійного оновлення знань та інструментів. Зрештою, виклики програмної інженерії полягають у балансуванні між вартістю розробки, швидкістю випуску та якістю продукту. Зменшення часу розробки мобільного додатка завдяки використанню інструментів CI/CD (Continuous Integration/ Continuous Delivery), що в свою чергу забезпечить швидкий та безболісний процес релізу додатку.

Контрольні запитання:

1. Що таке програмна інженерія?
2. Які основні цілі програмної інженерії?
3. Які проблеми вирішує програмна інженерія?

2. Аналіз та характеристика областей знань SWEBOK

Визначення SWEBOK

SWEBOK (Software Engineering Body of Knowledge) — це стандарт, який визначає ключові області знань у програмній інженерії. Він забезпечує систематизацію знань для кращого розуміння основних принципів і практик програмної інженерії.

Основні області знань SWEBOK

SWEBOK включає 15 основних областей, серед яких управління вимогами, проєктування, тестування, управління конфігурацією, якість ПЗ. Наприклад, управління вимогами охоплює аналіз і документування вимог до системи.

Застосування SWEBOK

SWEBOK використовується для стандартизації навчальних програм, професійної сертифікації та поліпшення процесів розробки ПЗ. Наприклад, він допомагає створювати навчальні курси для програмістів.

Контрольні запитання:

1. Що таке SWEBOK, і для чого він використовується?
2. Які ключові області знань включає SWEBOK?
3. Як SWEBOK впливає на розробку програмного забезпечення?
4. Які області знань є найважливішими для управління проєктами?

5. Чим відрізняються області знань SWEBOK, пов'язані з проєктуванням і тестуванням?

6. Як область управління вимогами допомагає покращити якість ПЗ?

7. Як SWEBOK допомагає у стандартизації процесів розробки ПЗ?

8. Яким чином SWEBOK підтримує навчальні програми з програмної інженерії?

9. У яких сферах професійної сертифікації застосовують SWEBOK?

3. Життєві цикли розробки програмного забезпечення (ЖЦ ПЗ)

ЖЦ ПЗ охоплює усі етапи створення програмного забезпечення: від аналізу вимог до його впровадження і підтримки. Ключова мета — створення ефективного ПЗ, що відповідає потребам замовника.

Основні моделі ЖЦ ПЗ:

Життєвий цикл програмного забезпечення (ЖЦ ПЗ) описує послідовність етапів створення програмного продукту: від аналізу вимог до підтримки після впровадження. Основні моделі ЖЦ включають: **водоспадну модель**, яка є послідовною і використовується для проєктів із чіткими, незмінними вимогами; **спіральну модель**, що підходить для проєктів із високим рівнем ризиків, завдяки ітеративному підходу; і **Agile** — гнучка модель, яка дозволяє адаптуватися до змін вимог на кожному етапі розробки. Кожна модель має свої переваги та недоліки. Водоспадна модель використовується для великих урядових проєктів із фіксованими вимогами. Agile підходить для розробки мобільних застосунків, де вимоги змінюються.

Порівняння моделей ЖЦ:

Кожна модель ЖЦ має свої особливості та підходить для різних типів проєктів. Водоспадна модель є простою у реалізації, але погано адаптується до змін у вимогах, що може бути проблемою для великих проєктів. Спіральна модель дозволяє працювати з високими ризиками та враховує зворотний зв'язок, але вимагає більших фінансових і часових витрат. Agile, зі свого боку, забезпечує швидкий випуск продукту, розбиваючи його на малі ітерації, але вимагає високого рівня залученості замовника. Наприклад, у створенні фінансових застосунків спіральна модель дозволяє ретельно протестувати всі етапи, тоді як Agile ідеально підходить для динамічних

ринків, таких як e-commerce.

Вибір моделі залежно від проєкту:

Вибір моделі ЖЦ залежить від таких факторів, як обсяг проєкту, рівень визначеності вимог, часові та фінансові обмеження. Для невеликих проєктів із чітко визначеними вимогами підходить водоспадна модель. Для великих та складних проєктів із високим рівнем ризиків, таких як створення медичних систем, рекомендується спіральна модель. Agile обирають для динамічних проєктів із частими змінами вимог, наприклад, у розробці мобільних додатків. Наприклад, при розробці CRM-системи для компанії з невеликою клієнтською базою доцільно використовувати водоспадну модель, тоді як для масштабного e-commerce проєкту Agile є ефективнішим.

Контрольні запитання:

1. Що таке життєвий цикл програмного забезпечення?
2. Які існують основні моделі ЖЦ ПЗ?
3. У чому відмінність між водоспадною моделлю та Agile?

4. Методи визначення вимог у програмній інженерії

Вимоги- це властивості, якими має володіти ПЗ для адекватного завдання функцій, а також умови та обмеження на ПЗ, дані, середовище виконання та техніку.

Вимоги відображають потреби людей (замовників, користувачів, розробників), зацікавлених у створенні програмного забезпечення. Замовник та розробник спільно проводять збір вимог, їх аналіз, перегляд, визначення необхідних обмежень та документування. Розрізняють вимоги до продукту та процесу, а також функціональні та нефункціональні вимоги, системні вимоги. Програмні вимоги визначають вимоги до процесу, ОС, режиму виконання, вибору платформи і т.п.

Етап збору вимог є одним із найважливіших у розробці ПЗ, оскільки він визначає, що саме має виконувати програма. Від якості зібраних вимог залежить успіх усього проєкту.

Типи вимог:

1. **Функціональні вимоги** описують, що система повинна робити, наприклад, виконання платежів або реєстрація користувачів (реєстрація користувачів).

2. **Нефункціональні вимоги стосуються характеристик роботи системи, таких як швидкість, безпека, продуктивність** (швидкість, безпека).

3. **Системні вимоги** визначають, як програмне забезпечення буде взаємодіяти з іншими системами

Наприклад, функціональна вимога для онлайн-магазину — можливість здійснення транзакцій, нефункціональна — забезпечення відповіді сервера менше ніж за 2 секунди.

Методи збору вимог:

Збір вимог є ключовим етапом у розробці програмного забезпечення, оскільки визначає його функціонал та очікувані результати. Найбільш поширеними методами збору вимог є **інтерв'ю** із замовниками для з'ясування їхніх потреб, **анкетування** для збору даних від широкої аудиторії, **воркшопи** для колективної розробки рішень, а також **спостереження** за тим, як користувачі взаємодіють із системою. Наприклад, при розробці ERP-системи для бізнесу воркшопи дозволяють залучити всіх ключових учасників процесу.

Аналіз та узгодження вимог із замовником:

Після збору вимог необхідно виконати їхній детальний аналіз і погодження із замовником. Аналіз дозволяє виявити суперечності, прогалини або дублювання вимог. Узгодження забезпечує впевненість, що кінцевий продукт відповідатиме очікуванням замовника. Наприклад, при створенні мобільного додатка для доставки їжі аналіз виявив дублювання функцій у модулі управління замовленнями, що дозволило оптимізувати розробку.

Інженерія вимог до ПЗ

Інженерія вимог – це процес збору, аналізу, документування та управління вимогами до програмного забезпечення. Вона включає ідентифікацію потреб замовників, формування функціональних і нефункціональних вимог та їх узгодження з усіма зацікавленими сторонами. Основні етапи: **еліцітація (збір вимог), аналіз, специфікація, верифікація та управління змінами**. Інженерія вимог допомагає уникнути непорозумінь між замовником і розробниками, зменшуючи ризики

невідповідності кінцевого продукту очікуванням користувачів.

Верифікація та формалізація вимог

Верифікація вимог – це процес перевірки їх повноти, коректності, несуперечливості та відповідності початковим цілям проєкту. Для цього використовуються методи, такі як **рецензування, прототипування, моделювання та аналіз сценаріїв використання**. Формалізація вимог означає представлення їх у структурованому вигляді за допомогою специфікацій, графічних моделей (UML, BPMN) або формальних мов (Z, Alloy). Це полегшує їх аналіз та подальшу реалізацію, особливо у великих і складних системах.

Контрольні запитання:

1. Які основні типи вимог існують?
2. Які методи використовуються для збору вимог?
3. Як забезпечити точність вимог до ПЗ?
4. Назвіть елементи об'єктно-орієнтованого моделювання програмних систем.
5. У чому полягає принцип приховування інформації?
6. Визначте концепцію моделі сценаріїв для збирання вимог.
7. Дайте пояснення для нотації діаграми сценаріїв та базових взаємин у них.
8. Назвіть основні типи об'єктів моделі.
9. Наведіть завдання трасування вимог.
10. Розкажіть про принципи взаємовідносин між замовником та розробником вимог до системи.

5. Методи аналізу і побудування моделей програмних об’єктів (ПрО)

Аналіз і моделювання програмних об’єктів (ПрО) є важливими етапами в розробці програмного забезпечення, що дозволяють зрозуміти структуру, поведінку та взаємодію компонентів системи. Методи аналізу допомагають розробникам визначити вимоги до системи та спроєктувати її таким чином, щоб вона відповідала функціональним і нефункціональним вимогам.

Моделювання програмних об'єктів здійснюється за допомогою різних підходів, включаючи **об’єктно-орієнтований аналіз (ООА), функціональний аналіз, структурне моделювання та моделювання поведінки системи**

Методи аналізу програмних об'єктів

Методи аналізу ПрО дозволяють формалізувати структуру та взаємозв’язки між компонентами системи. Основні методи включають:

Об'єктно-орієнтований аналіз (ООА)

Об'єктно-орієнтований аналіз базується на представленні системи як набору об'єктів, що взаємодіють між собою. Він включає:

1. Визначення **класів і об’єктів**, їх атрибутів та методів.
2. Виявлення **зв’язків між класами**, включаючи асоціацію, композицію, наслідування.
3. Аналіз сценаріїв використання (Use Case) для визначення взаємодії між об’єктами.

Приклад: При розробці онлайн-магазину можна виділити такі об'єкти: "Користувач", "Товар", "Замовлення", які матимуть відповідні атрибути (ім’я, ціна, статус) та методи (оформити замовлення, скасувати покупку).

Функціональний аналіз

Функціональний аналіз зосереджується на визначенні функцій, які виконує система. Основні методи:

1. **Діаграми потоків даних (DFD)** – графічне представлення руху інформації в системі.
2. **Функціональна декомпозиція** – поділ складної системи на підсистеми та окремі функції.
3. **Метод чорної скриньки** – розгляд системи з точки зору вхідних і вихідних даних без аналізу її внутрішньої реалізації.

Приклад: У банківській системі DFD-діаграма може показати, як дані про клієнта проходять через процес "Оформлення кредиту", включаючи перевірку платоспроможності та збереження даних у базі.

Аналіз потоків управління

Методика аналізу потоків управління фокусується на розподілі процесів у системі та визначенні логіки виконання операцій.

1. **Діаграми активності UML** дозволяють моделювати алгоритми бізнес-логіки системи.

2. **Графи потоку керування (Control Flow Graph)** використовуються для аналізу логічних гілок у програмному коді.

Методи побудови моделей програмних об'єктів

Моделі ПрО використовуються для графічного представлення структури, логіки та поведінки системи. Основні методи моделювання включають:

Структурне моделювання

Структурне моделювання фокусується на організації компонентів системи.

1. **Діаграми класів UML** відображають об'єкти, їх атрибути, методи та взаємозв'язки.
2. **Діаграми компонентів** описують взаємодію між модулями ПЗ.

Поведінкове моделювання

Поведінкове моделювання описує, як система змінює свій стан у відповідь на події.

1. **Діаграми послідовностей** показують взаємодію між об'єктами у хронологічному порядку.
2. **Діаграми станів** описують зміну станів об'єкта впродовж життєвого циклу.

Використання UML для моделювання:

Уніфікована мова моделювання (UML) є стандартом для опису архітектури програмного забезпечення. Вона використовується для створення діаграм, які відображають структуру і поведінку системи. UML діаграми дозволяють краще зрозуміти взаємодію компонентів ще до початку розробки. Наприклад, при створенні системи бізнес аналізу для туристів, ми можемо виділити основні сценарії поведінки користувача: створення запиту на аналіз, перегляд отриманої візуалізації даних, створення коментаря. Всі ці сценарії можна відобразити на діаграмі прецедентів на рисунку 1.

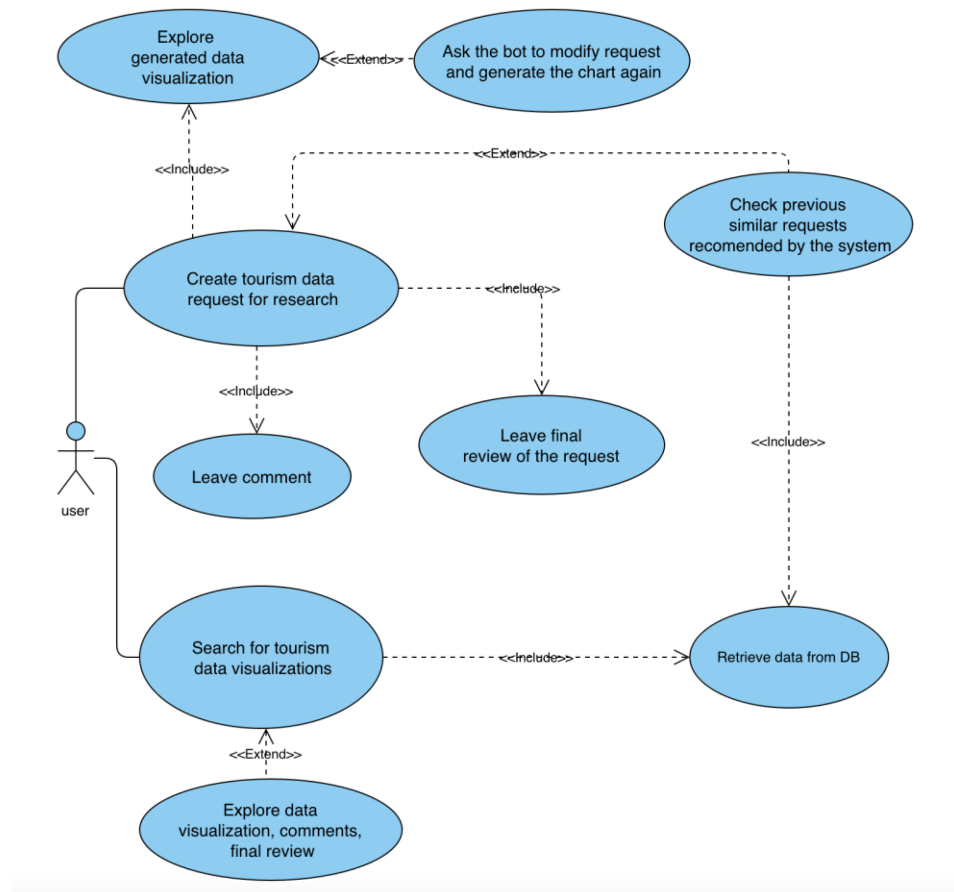

Рисунок 5.1 - Діаграма прецедентів

Об’єктно-орієнтоване моделювання

Цей підхід поєднує структурні та поведінкові аспекти системи.

1. **Об'єктні моделі** описують класи, їхні атрибути та методи.
2. **Компонентні моделі** описують інтеграцію модулів.

Приклад: У веб застосунку модель може містити класи "Користувач", "Адміністратор", "Сесія", кожен з яких має певні функції та методи.

Контрольні питання та завдання

1. Визначте завдання аналізу предметної галузі та процесів проектування архітектури системи.
2. Сформулюйте задачі концептуального проектування моделей ПрВ.
3. Назвіть продукти аналізу домену у методі Шлаєр та Меллора.
4. Назвіть моделі методу Шлаєр та Меллора та їх суть. 5.Які ще моделі ПрО Ви знаєте?

1. Перерахуйте ключові чинники, що впливають проектування інтерфейсів.
2. Назвіть приклади нефункціональних вимог, які слід враховувати на стадії проектування архітектури.

3. Які рівні виділяються в архітектурі системи?

4. Які відомі способи об'єднання об'єктів у підсистеми?

5. Назвіть прийоми забезпечення перенесення системи до іншого середовища.

6. Методи проєктування програмних систем

Проєктування програмних систем визначає, як буде організоване ПЗ на рівні архітектури, модулів та їх взаємодії, забезпечуючи масштабованість і простоту підтримки.

Архітектурні патерни:

Архітектурні патерни — це готові рішення для організації структури програмного забезпечення, які допомагають ефективно впоратися із задачами проєктування. Наприклад, патерн MVC (Model-View-Controller) розділяє дані, інтерфейс користувача та логіку додатка, що спрощує підтримку та розвиток. Патерн Microservices підходить для великих систем, де окремі компоненти працюють як автономні сервіси. Layered Architecture використовується для поділу системи на рівні (презентації, логіки, даних), забезпечуючи чітку структуру. Наприклад, у мобільному банкінгу патерн MVC дозволяє відокремити інтерфейс для користувачів від процесів обробки транзакцій.

Проєктування інтерфейсів API:

API (Application Programming Interface) — це інтерфейси, які дозволяють різним програмам взаємодіяти між собою. Ефективне проєктування API включає чіткий опис методів, форматів даних (наприклад, JSON або XML) та забезпечення безпеки. API повинні бути зрозумілими, гнучкими та добре задокументованими. Наприклад, REST API використовується у веброзробці для забезпечення зв'язку між сервером і клієнтом, де методи GET, POST, PUT і DELETE обробляють відповідні запити.

Інструменти для проєктування:

Інструменти для проєктування використовуються для створення архітектурних схем і діаграм, які спрощують комунікацію між учасниками проєкту. Visio дозволяє створювати діаграми процесів, схеми мереж і блок-схеми. Draw.io — зручний

інструмент для моделювання систем, доступний онлайн. Enterprise Architect використовується для моделювання складних систем із використанням UML. Наприклад, за допомогою Draw.io можна створити схему бази даних для e-commerce платформи.

Компонентне проєктування

Компонентне проєктування базується на створенні незалежних модулів, які можуть бути повторно використані в різних системах.

Архітектура на основі компонентів

Система складається з незалежних модулів, які взаємодіють через чітко визначені інтерфейси.

Використовуються інтерфейси API для взаємодії між компонентами.

Приклад: У корпоративних системах модулі "Облік клієнтів", "Фінансові операції" та "Логістика" можуть бути реалізовані окремими сервісами.

Контейнери компонентів

Використання Spring Framework, JavaBeans, COM/DCOM для реалізації компонентної архітектури.

Приклад: У вебсервісах застосовується Spring Boot для побудови модульних мікросервісів.

Сервісно-орієнтоване проєктування (SOA – Service-Oriented Architecture)

Цей підхід передбачає розбиття системи на сервіси, які взаємодіють через мережу за допомогою відкритих стандартів.

Основні концепції SOA

Всі функції системи реалізовані у вигляді незалежних сервісів.

Використання REST API або SOAP для комунікації між сервісами.

Приклад: У фінансових системах мікросервіс "Авторизація платежів" може працювати окремо від сервісу "Обробка запитів клієнтів".

Контейнери для управління сервісами

Використання Docker та Kubernetes для розгортання сервісів.

Приклад: У хмарних системах кожен сервіс може бути запущений в окремому контейнері, що забезпечує гнучкість та масштабованість.

Аспектно-орієнтоване програмування (Aspect-Oriented Programming, AOP)

Визначення та суть аспектно-орієнтованого програмування

Аспектно-орієнтоване програмування (AOP) – це парадигма програмування, яка дозволяє відокремлювати перехресні (нелокальні) аспекти поведінки програми, які важко або неможливо інкапсулювати в окремих класах у межах об'єктно-орієнтованого програмування (ООП).

Перехресні аспекти – це функціональність, яка повторюється в різних частинах програми та не відноситься до основної логіки бізнес-процесу. Наприклад, логування, безпека, транзакції, обробка помилок.

AOP дозволяє виділити такі аспекти у незалежні модулі, які можуть бути динамічно впроваджені у виконуваний код без зміни його основної структури.

Аспекти (Aspects)

Аспекти – це окремі модулі, що містять код, який реалізує певну функціональність, незалежну від основної логіки програми.
Приклад: Аспект логування може записувати в журнал усі виклики методів, незалежно від класу, в якому вони виконуються.

Точки з’єднання (Join Points)

Точка з’єднання – це місце в коді, де може бути впроваджений аспект (наприклад, виклик методу, створення об'єкта, виконання блоку коду).

Впровадження (Advice)

Advice – це код, який виконується у певний момент часу відносно точки з’єднання. Існують три основні види Advice:

1 Before advice – виконується перед викликом методу.

2 After advice – виконується після виклику методу.

3 Around advice – обгортає метод і дозволяє змінювати його поведінку.

Приклад: Before advice може перевіряти права доступу перед викликом певного методу.

Правила впровадження (Pointcuts)

Pointcut – це набір правил, які визначають, коли саме потрібно впроваджувати аспект у код.

Приклад: Виконання логування перед викликами всіх методів у класах, що починаються на "Service".

Вплив на код (Weaving)

Weaving – це процес "вплітання" аспектів у вихідний код на різних етапах виконання програми:

Compile-time weaving – впровадження аспектів під час компіляції.

Load-time weaving – впровадження аспектів під час завантаження класів.

Runtime weaving – впровадження аспектів під час виконання програми (динамічне впровадження).

Використання AOP у програмуванні

Логування (Logging)

AOP дозволяє реалізувати централізоване логування без зміни вихідного коду.

```
@Aspect
public class LoggingAspect {
    @Before("execution(* com.example.service.*.*(..))")
    public void logBeforeMethod(JoinPoint joinPoint) {
        System.out.println("Logging before method: " +
joinPoint.getSignature().getName());
    }
}
```

Цей аспект логуватиме всі методи в пакеті com.example.service.

Управління транзакціями

AOP дозволяє автоматично керувати транзакціями, не змінюючи бізнес-логіку.

```
@Aspect
public class LoggingAspect {
    @Before("execution(* com.example.service.*.*(..))")
    public void logBeforeMethod(JoinPoint joinPoint) {
        System.out.println("Logging before method: " +
joinPoint.getSignature().getName());
    }
}
```

Цей аспект забезпечує початок і завершення транзакції навколо виклику методів у DAO-рівні.

Контрольні запитання:

1 Що таке архітектурний патерн?

2 Які переваги має проєктування за патернами?

3 Як спроєктувати ефективний API?

4 Об'єкти генеруючого програмування та коротка їх характеристика.

7.Інженерія ПВК (програмних відтворюваних компонентів)

Інженерія програмних відтворюваних компонентів (ПВК) спрямована на створення багаторазових компонентів, які можуть бути використані в різних програмних системах.

Основні принципи інженерії ПВК

1. **Модульність** – кожен компонент повинен мати чітко визначений функціонал.

2. **Повторне використання** – можливість інтеграції компонентів у різні програмні системи.

3. **Інкапсуляція** – приховування внутрішньої реалізації та надання доступу лише через інтерфейси.

4. **Гнучкість та адаптивність** – можливість налаштування параметрів компонентів під різні завдання.

Приклад: У розробці CRM-системи можна використовувати багаторазові компоненти "Аутентифікація", "Керування користувачами", "Генерація звітів", які можуть бути інтегровані в інші програми.

Різновиди ПВК

Програмні відтворювані компоненти можна класифікувати за різними критеріями:

За рівнем абстракції

1. **Функціональні компоненти** – реалізують певну функцію (наприклад, "Обробка платежів", "Фільтрація даних").

2. **Інфраструктурні компоненти** – забезпечують підтримку роботи системи (наприклад, "Журналювання помилок", "Кешування даних").

За способом інтеграції

1. **Бібліотечні компоненти** – підключаються до програм як стандартні бібліотеки.

2. **Сервісні компоненти** – реалізовані у вигляді незалежних мікросервісів, що взаємодіють через API.

За технологіями реалізації

1. **Об’єктно-орієнтовані компоненти** – використовують ООП-концепції для повторного використання коду.

2. **Компоненти на основі XML/JSON** – застосовуються для обміну даними між системами.

3. **Компоненти на основі хмарних технологій** – працюють у вигляді сервісів (наприклад, AWS Lambda, Google Cloud Functions).

Приклад: Веб застосунок може містити функціональний компонент "Реєстрація користувачів", інфраструктурний компонент "Моніторинг активності" та сервісний компонент "Платіжний шлюз".

Специфікація ПВК

Специфікація ПВК – це формальний опис компонентів, що визначає їхню функціональність, інтерфейси, вимоги до використання та залежності.

Основні елементи специфікації

1. **Опис функціональності** – що виконує компонент.

2. **Інтерфейси взаємодії** – способи підключення та взаємодії з іншими модулями.

3. **Вимоги до середовища виконання** – апаратні та програмні залежності.

4. **Політика безпеки** – рівень доступу, обмеження, шифрування даних.

Документування ПВК

1. **Формальний опис** – XML, JSON, UML-діаграми.

2. **Технічна документація** – специфікації API, приклади використання, інструкції.

3. **Приклад коду** – приклади реалізації та тестових сценаріїв.

Приклад: Компонент "Обробка платежів" у фінансовій системі має API-специфікацію, що визначає, як відправляти запити на оплату, як отримувати статуси транзакцій та які рівні доступу потрібні.

Репозитарій компонентів

Репозитарій компонентів – це централізоване сховище багаторазових модулів, яке забезпечує їхнє зберігання, управління версіями та повторне використання.

Типи репозитаріїв

1. **Локальні репозитарії** – зберігаються у внутрішній мережі компанії.
2. **Публічні репозитарії** – відкриті платформи, такі як GitHub, Maven, npm, Docker Hub.

Управління версіями компонентів

1. **Семантичне версіонування** (Semantic Versioning) – використання форматів Major.Minor.Patch (наприклад, 1.2.3).
2. **Забезпечення зворотної сумісності** – перевірка взаємодії нових версій компонентів із наявною системою.

Автоматизація використання компонентів

1. **Пакетні менеджери**: Maven (Java), npm (JavaScript), PyPI (Python).
2. **CI/CD-пайплайни** для автоматичного оновлення компонентів.

Приклад: Використання **Docker Hub** для збереження контейнеризованих мікросервісів, що дозволяє легко розгортати готові компоненти в різних середовищах.

Інженерія додатків та предметної області

Інженерія додатків та інженерія предметної області взаємопов’язані:

1. Інженерія предметної області аналізує сферу застосування ПЗ, визначає стандарти та готує багаторазові компоненти.
2. Інженерія додатків використовує ці компоненти для створення конкретних програмних продуктів.

Процес адаптації ПВК для конкретної предметної області

1. Визначення **специфічних вимог** для певного бізнес-сегмента.
2. Використання **готових компонентів** із репозитарію.
3. Інтеграція компонентів у конкретну систему та **тестування** їхньої роботи.

Приклад: У медицині можна створити універсальні модулі "Запис на прийом", "Збереження історії хвороб", які адаптуються до різних клінік.

Контрольні запитання:

1. Що таке програмний відтворюваний компонент (ПВК)?
2. Які основні принципи інженерії ПВК?
3. Як модульність впливає на повторне використання компонентів?

4. Чому важлива інкапсуляція при розробці ПВК?
5. Наведіть приклади програмних компонентів, які можуть використовуватися в різних системах.

8. Методи тестування і верифікації ПЗ

Тестування та верифікація забезпечують якість програмного забезпечення шляхом пошуку помилок і перевірки відповідності початковим вимогам.

Методи доказу правильності програм

Доказ правильності програм є важливою складовою формального аналізу програмного забезпечення. Він дозволяє підтвердити, що програма виконує свої функції відповідно до заданих специфікацій, без логічних помилок і непередбачуваної поведінки.

Формальні методи доказу правильності використовуються в критичних системах, де навіть невелика помилка може призвести до серйозних наслідків (наприклад, у медичних, банківських та авіаційних системах). Вони засновані на використанні математичних моделей для аналізу програмного коду.

Загальна характеристика формальних методів доказу

Формальні методи доказу правильності програм застосовують математичні підходи для підтвердження відповідності програм їхнім специфікаціям. Вони забезпечують:

1. **Повну перевірку логічної коректності** без необхідності виконання програми.
2. **Гарантоване уникнення деяких класів помилок**, які можуть не бути виявлені під час тестування.

Формальні методи включають:

1. **Аксіоматичний підхід** – використання логічних тверджень та інваріантів для опису поведінки програми.
2. **Алгебраїчні методи** – представлення програм у вигляді математичних рівнянь і їх аналіз.
3. **Символьне виконання** – аналіз виконання програми без конкретних значень змінних.
4. **Перегляд структури програми** – аналіз її схеми та логіки роботи.

Приклад: Для перевірки коректності алгоритму сортування можна використати логічний підхід, де перевіряється, що кожен елемент після виконання алгоритму знаходиться у правильному порядку.

Техніка символьного виконання

Символьне виконання (Symbolic Execution) – це метод, що дозволяє перевіряти програмний код без його реального виконання. Замість конкретних значень змінних використовуються символи або математичні вирази.

Основні принципи символьного виконання

1. Програма виконується "в уявному середовищі", де змінні представлені не числами, а символами.
2. Формуються **умовні вирази**, які відображають можливі стани програми.
3. Для кожного можливого шляху виконання формується система рівнянь, яка перевіряється на логічну коректність.

Методи перегляду структури програми

Методи перегляду структури програми базуються на аналізі її побудови та логіки виконання, що дозволяє перевірити її правильність ще до тестування.

Аналіз потоку управління

Аналізуються всі можливі шляхи виконання програми з точки зору:

1. Взаємодії між функціями та модулями.
2. Наявності "мертвого коду" (невиконуваних гілок).
3. Ризиків зациклення та рекурсивного виклику.

Приклад: Якщо певний блок коду ніколи не виконується через помилкову умову (`if (false)`), це можна виявити під час статичного аналізу.

Аналіз потоку даних

Цей метод дозволяє перевірити, як змінні ініціалізуються та використовуються в програмі. Основні аспекти:

1. Визначення змінних, які можуть бути використані без ініціалізації.
2. Аналіз потенційних конфліктів при доступі до спільних ресурсів.
3. Виявлення зайвих або некоректних змінних.

Приклад: У програмі `int a; if (b > 0) a = 5; print(a);` може виникнути помилка, якщо `b <= 0`, оскільки змінна `a` не була ініціалізована.

Використання графів потоку керування

1. Будується **граф потоку керування** (Control Flow Graph, CFG), що відображає всі можливі переходи між блоками коду.

2. Використовується для аналізу циклів, перевірки можливості виходу за межі функції та виявлення помилок у переходах між станами.

Приклад: Для програми, що містить кілька вкладених умов, граф потоку керування допомагає оцінити, які умови є критичними для коректного виконання.

Типи тестування:

Тестування ПЗ включає різні типи, кожен із яких виконує свою роль. **Юніт-тестування** перевіряє окремі модулі або функції на рівні коду. **Інтеграційне тестування** перевіряє взаємодію між компонентами. **Системне тестування** оцінює роботу програми в цілому, включаючи функціонал, продуктивність і безпеку. Наприклад, у розробці CRM-системи юніт-тести перевіряють обробку заявок, а системне тестування забезпечує перевірку роботи всієї платформи під час високих навантажень.

Автоматизоване тестування:

Автоматизоване тестування спрощує процес перевірки ПЗ, зменшує людський фактор і скорочує час. Інструменти, такі як **Selenium**, використовуються для тестування вебз астосунків, тоді як **JUnit** ідеально підходить для автоматизації тестування коду на Java. Наприклад, Selenium дозволяє створити сценарії для автоматичної перевірки функцій інтернет-магазину, таких як пошук товарів і оформлення замовлень.

Методи верифікації:

Верифікація ПЗ — це процес перевірки того, чи відповідає продукт початковим вимогам. Методи верифікації включають **код-рев'ю** (перевірку коду іншими розробниками), **аналіз вимог** (оцінка відповідності результатів очікуванням замовника) і **проведення тестів вручну**. Наприклад, код-рев'ю допомагає виявити потенційні проблеми в алгоритмах перед впровадженням змін у продукт.

Контрольні запитання:

1. Які типи тестування існують?
2. Чим відрізняється тестування від верифікації?
3. Які інструменти використовують для автоматизованого тестування?

4. У чому полягає відмінність техніки формального доказу від символьного виконання програм?
5. Сформулюйте основні завдання верифікації та валідації програм.
6. У чому відмінність верифікації та валідації?
7. Визначте процес тестування.
8. Назвіть методи тестування.

9. Інтеграція та змінення компонентів ПЗ

Процес інтеграції дозволяє об'єднати різні компоненти ПЗ в єдину систему, а також забезпечити її адаптацію до нових вимог.

Інтеграційні платформи:

Інтеграційні платформи дозволяють автоматизувати процеси інтеграції та доставки компонентів ПЗ. **Jenkins** і **GitLab CI/CD** широко використовуються для налаштування конвеєрів розробки, які включають тестування, збірку та розгортання коду. Наприклад, Jenkins дозволяє автоматично протестувати новий модуль, зібрати його в контейнер і розгорнути на сервері.

Управління змінами в ПЗ:

Управління змінами є важливою складовою життєвого циклу ПЗ. Це включає документування змін, контроль версій і забезпечення узгодженості між новими й існуючими функціями. Системи, такі як **Git**, допомагають відслідковувати зміни в коді та координувати роботу команд. Наприклад, додавання нового модуля у фінансову систему проводиться поетапно, щоб уникнути порушення роботи наявного функціоналу.

Оцінка ризиків під час інтеграції:

Оцінка ризиків під час інтеграції включає аналіз можливих проблем, які можуть виникнути під час об'єднання компонентів системи. Наприклад, ризики можуть бути пов'язані з несумісністю API або конфліктами даних у базах. Для їх мінімізації проводять тестування інтеграцій і створюють резервні копії даних. Наприклад, перед інтеграцією нового модуля в e-commerce платформу проводиться перевірка сумісності всіх залежностей.

Контрольні запитання:

1. Що таке інтеграція компонентів ПЗ?
2. Як правильно керувати змінами в програмному забезпеченні?
3. Які інструменти спрощують інтеграцію компонентів?

10.Реінженерія програмних систем

Визначення реінженерії

Реінженерія програмних систем – це процес аналізу, реструктуризації та вдосконалення існуючого програмного забезпечення з метою підвищення його продуктивності, гнучкості, безпеки та масштабованості. Це включає **розширення функціоналу, виправлення застарілих частин коду, міграцію на нові технології та архітектурні зміни**.

Основні етапи реінженерії

1. **Аналіз наявної системи** – виявлення її сильних і слабких сторін.
2. **Виділення основних компонентів** – виявлення ключових модулів, що потребують змін.
3. **Рефакторинг коду** – покращення якості коду без зміни його функціональності.
4. **Оновлення архітектури** – перехід до більш сучасних підходів, наприклад, від моноліту до мікросервісів.
5. **Міграція даних** – адаптація баз даних до нових вимог.
6. **Тестування та верифікація** – перевірка відповідності нової версії системи початковим бізнес-вимогам.

Переваги реінженерії

Підвищення продуктивності та швидкодії системи. Зниження витрат на підтримку та експлуатацію. Зменшення технологічного боргу та оновлення старих технологій. Поліпшення безпеки та відповідності новим стандартам.

Приклад: Реінженерія банківської системи, що працює на застарілому мейнфреймі, шляхом перенесення її в хмарну архітектуру.

Рефакторинг компонентів

Визначення рефакторингу

Рефакторинг – це процес **покращення внутрішньої структури коду без зміни його зовнішньої поведінки**. Це необхідно для зменшення складності, підвищення зручності супроводу та зниження ризику помилок.

Основні методи рефакторингу

1. **Розбиття великих класів на дрібніші (Extract Class)** – підвищує модульність коду.
2. **Видалення дублювання коду (Remove Duplication)** – усуває повторювані ділянки.
3. **Зміна довгих методів на короткі (Extract Method)** – покращує читабельність.
4. **Перейменування змінних та методів (Rename Variable, Rename Method)** – робить код зрозумілішим.
5. **Заміна умовних операторів на поліморфізм (Replace Conditional with Polymorphism)** – робить код більш гнучким.

Інструменти для рефакторингу

IntelliJ IDEA, Eclipse, Visual Studio – вбудовані функції автоматичного рефакторингу.

SonarQube – аналіз якості коду та виявлення проблемних місць.

Переваги рефакторингу

1. Покращує читабельність коду.
2. Зменшує ризик помилок у майбутньому.
3. Полегшує розширення та модифікацію системи.

Приклад: Рефакторинг великого монолітного додатка шляхом виділення бізнес-логіки у окремі мікросервіси.

Реверсна інженерія

Визначення реверсної інженерії

Реверсна інженерія – це процес **аналізу та відтворення структури, логіки та функціональності існуючого програмного забезпечення**. Вона використовується для **відновлення документації, виявлення вразливостей та адаптації коду** до нових потреб.

Основні етапи реверсної інженерії

1. **Декомпіляція коду** – перетворення виконуваного файлу назад у вихідний

код (декомпілятори: JD-GUI, JADX).

2. **Аналіз архітектури** – побудова UML-діаграм для розуміння структури програми.

3. **Відновлення документації** – формування специфікацій і опису бізнес-логіки.

4. **Пошук вразливостей** – аналіз безпеки для усунення потенційних загроз.

Застосування реверсної інженерії

1. **Модернізація застарілих програм** – відновлення коду для оновлення або перенесення на нові технології.

2. **Аналіз шкідливого ПЗ** – виявлення способів роботи вірусів і хакерських атак.

3. **Перевірка наявного ПЗ** – аналіз сторонніх програм перед інтеграцією в систему.

Інструменти для реверсної інженерії

1. **IDA Pro, Ghidra** – аналіз двійкових файлів.
2. **JD-GUI, JADX** – декомпіляція Java-додатків.
3. **Radare2, OllyDbg** – інструменти для дослідження виконуваного коду.

Контрольні запитання:

1. Визначте основні завдання реінженерії.
2. Визначте основні операції рефакторингу компонентів.
3. Визначте основні операції реінженерії програмних систем.

9.Моделі якості та надійності у програмній інженерії

Якість програмного забезпечення (ПЗ) є ключовим фактором його успіху та прийнятності для користувачів. Моделі якості дозволяють визначити, оцінити та контролювати рівень якості ПЗ, а моделі надійності допомагають оцінити стабільність та безвідмовність роботи системи.

Основними аспектами оцінки якості є: Функціональна коректність – наскільки система відповідає вимогам. Продуктивність – ефективність використання ресурсів. Надійність – здатність функціонувати без помилок протягом заданого часу.

Безпека – захист від несанкціонованого доступу та атак.

Зручність використання – наскільки ПЗ є зрозумілим і простим у використанні.

Основні моделі якості ПЗ

Існує кілька підходів до моделювання якості ПЗ:

1. **Модель ISO/IEC 25010** – визначає якісні характеристики, такі як функціональність, ефективність, зручність використання, надійність, безпека, сумісність, підтримуваність та портативність.
2. **Модель Гарвера-Рома (McCall's Model)** – зосереджується на внутрішніх та зовнішніх якостях ПЗ, таких як зручність модифікації та тестованість.
3. **Модель Dromey** – пропонує структурний підхід до визначення якості, розділяючи її на атрибути вимог, конструкції та реалізації.

Метрики якості програмного забезпечення

Метрики якості використовуються для вимірювання різних характеристик ПЗ:

1. **Метрики коду** – довжина коду, складність за цикломатичною мірою Маккейба.
2. **Метрики надійності** – середній час між відмовами (MTBF), середній час відновлення (MTTR).
3. **Метрики продуктивності** – швидкість обробки запитів, використання пам'яті та процесорного часу.
4. **Метрики підтримуваності** – оцінка складності коду, рівень повторного використання компонентів.

Приклад: Якщо програма містить багато умовних операторів if-else, її цикломатична складність зростає, що ускладнює тестування.

Стандартний метод оцінки значень показників якості

Стандартизовані підходи до оцінки якості ПЗ включають:

1. **Методи експертного оцінювання** – суб'єктивна оцінка спеціалістами.
2. **Формальні моделі** – застосування математичних методів для визначення показників якості.
3. **Аналіз дефектів** – виявлення та класифікація помилок у коді.
4. **Методи тестування** – функціональне тестування, навантажувальне тестування тощо.

Управління якістю програмних систем

Управління якістю ПЗ передбачає:

1. **Планування якості** – визначення стандартів і процедур контролю якості.
2. **Забезпечення якості (QA – Quality Assurance)** – процеси, що гарантують відповідність стандартам.
3. **Контроль якості (QC – Quality Control)** – виявлення дефектів та їх усунення.

Приклад: Використання автоматизованих інструментів (наприклад, SonarQube) для аналізу якості коду допомагає підтримувати високий рівень стандартів.

Модель якості ПЗ

Модель якості ПЗ визначає основні характеристики, які впливають на загальну якість програмного продукту.

Основні складові моделі якості

1. **Функціональність** – наскільки система відповідає вимогам користувача.
2. **Надійність** – здатність програми працювати без збоїв.
3. **Продуктивність** – ефективність використання ресурсів.
4. **Зручність використання** – наскільки система зрозуміла для користувача.
5. **Безпека** – захист даних та користувачів.
6. **Сумісність** – взаємодія з іншими системами.
7. **Підтримуваність** – легкість внесення змін.
8. **Переносимість** – здатність працювати на різних платформах.

Приклад оцінки якості ПЗ

Використання **ISO/IEC 25010**, що включає якісні атрибути:

1. **Correctness** – правильність виконання функцій.
2. **Efficiency** – ефективність використання ресурсів.
3. **Reliability** – стійкість до відмов.

Приклад: Оцінка зручності використання мобільного додатку шляхом A/B тестування різних версій інтерфейсу.

Метрики якості програмного забезпечення

Види метрик якості

1. **Метрики коду**:

1.1. Кількість рядків коду (LOC – Lines of Code).

1.2. Цикломатична складність (McCabe's Complexity).

1.3. Кількість коментарів (Comment Ratio).

2. **Метрики продуктивності**:

2.1. Час відгуку системи (Response Time).

2.2. Використання пам'яті (Memory Usage).

3. **Метрики тестованості**:

3.1. Співвідношення помилок до рядків коду.

3.2. Відсоток покриття коду тестами (Code Coverage).

4. **Метрики надійності**:

4.1. Середній час між відмовами (MTBF).

4.2. Середній час відновлення (MTTR).

Приклад: Якщо система має низький показник MTBF, необхідно покращити тестування та усунути слабкі місця в коді.

Стандартний метод оцінки значень показників якості

Оцінка якості ПЗ проводиться за такими етапами:

1. **Збір даних** – аналіз вимог, збір метрик.
2. **Аналіз** – застосування математичних моделей для обробки даних.
3. **Валідація** – перевірка відповідності отриманих результатів стандартам.
4. **Оптимізація** – внесення змін для покращення показників якості.

Приклад: Використання тестування навантаження для оцінки продуктивності веб сервісу під високим трафіком.

Управління якістю ПЗ

Процеси управління якістю

1. **Планування якості** – визначення стандартів та політик.
2. **Гарантія якості (QA)** – запобігання дефектам на етапах розробки.
3. **Контроль якості (QC)** – виявлення та виправлення помилок.

Інструменти управління якістю

1. **Статичний аналіз коду** – SonarQube, Checkstyle.
2. **Автоматизоване тестування** – Selenium, JUnit.
3. **Моніторинг продуктивності** – New Relic, Datadog.

Приклад: Використання DevOps-підходу для автоматичного розгортання та

тестування коду в CI/CD.

Контрольні питання та завдання

1. Визначте поняття – якість ПЗ.
2. Назвіть основні аспекти та рівні моделі якості ПЗ.
3. Визначте характеристики якості ПЗ та їх призначення.
4. Які методи використовуються щодо показників якості?
5. Визначте метрики програмного продукту та їх складові.
6. Які стандарти у сфері якості ПЗ існують?

11. Методи управління проєктом, ризиком і конфігурацією

Управління проєктами у сфері програмної інженерії передбачає **координацію ресурсів, контроль якості, управління ризиками та конфігурацією програмного забезпечення**. Це необхідно для успішної реалізації проєкту, дотримання строків та бюджетних обмежень.

Методи управління проєктами

Управління проєктами – це процес **організації, планування та контролю всіх етапів життєвого циклу проєкту**, починаючи від аналізу вимог до підтримки кінцевого продукту.

Основні завдання управління проєктами

1. Планування та складання графіків виконання проєкту.
2. Організація робочих груп та координація команди.
3. Управління ризиками та оцінка потенційних загроз.
4. Контроль виконання робіт та досягнення поставлених цілей.
5. Забезпечення якості продукту та керування конфігурацією системи.

Основні методи управління проєктами

1. **Класичний (водоспадний) метод** – реалізація проєкту через чітко визначені послідовні етапи.
2. **Гнучкі методології (Agile, Scrum, Kanban)** – адаптивний підхід з ітеративною розробкою та частими випусками продукту.

3. **Метод критичного шляху (CPM – Critical Path Method)** – розрахунок найбільш тривалих процесів для визначення мінімального строку реалізації проєкту.

4. **PERT (Program Evaluation and Review Technique)** – метод оцінки та перегляду прогресу проєкту за допомогою ймовірнісних оцінок часу виконання завдань.

Приклад: Якщо в програмному проєкті є критичні завдання, які не можна перенести (наприклад, тестування безпеки), застосовується CPM для їх визначення та контролю.

Методи управління програмним проєктом

Управління програмним проєктом враховує **специфічні особливості розробки ПЗ**, такі як складність програмного коду, інтеграція з іншими системами та потреба в регулярному тестуванні.

Етапи управління програмним проєктом

1. **Аналіз та визначення вимог** – розробка документації, узгодження специфікацій.

2. **Планування проєкту** – розробка графіка робіт, розподіл ресурсів.

3. **Розробка та тестування** – написання коду, його перевірка, виявлення та усунення помилок.

4. **Реліз та супровід** – випуск продукту, оновлення та підтримка.

Приклад: Використання Scrum у розробці мобільного додатку дозволяє розбити процес на короткі ітерації (спринти) та гнучко адаптувати функціональність під зміни користувацьких потреб.

Планування проєкту

Планування є **одним із найважливіших етапів управління програмними проєктами**. Воно дозволяє оцінити ресурси, розподілити завдання та сформувати чіткий графік реалізації.

Основні компоненти плану проєкту

1. **Графік виконання робіт** (CPM, PERT).

2. **Оцінка ресурсів та бюджету**.

3. **Управління ризиками** – передбачення можливих проблем.

4. **Аналіз залежностей між задачами**.

Приклад: Планування програмного проєкту із застосуванням діаграм Ганта

дозволяє чітко відслідковувати прогрес виконання завдань та відповідальних за них співробітників.

Організаційні аспекти управління у проєкті

Ролі учасників проєкту

1. **Менеджер проєкту** – контролює виконання плану та ресурсів.
2. **Аналітики** – визначають вимоги до системи.
3. **Розробники** – створюють програмний код.
4. **Тестувальники** – перевіряють якість продукту.
5. **Замовник** – визначає бізнес-вимоги та оцінює кінцевий продукт.

Ефективна комунікація в команді

1. Використання Agile-методологій для гнучкої взаємодії.
2. Регулярні зустрічі команди (daily meetings).
3. Використання трекерів завдань (Jira, Trello).

Оцінювання проєкту

Правильна оцінка вартості та строків проєкту дозволяє **уникнути перевитрат ресурсів та зриву дедлайнів**.

Методи оцінки програмного проєкту

1. **Метод експертних оцінок** – використання думок досвідчених спеціалістів.
2. **Алгоритмічні моделі** (COCOMO) – оцінка вартості розробки на основі розміру коду.
3. **Метод аналогій** – порівняння з попередніми проєктами.

Приклад: Модель COCOMO використовується для оцінки тривалості розробки ПЗ на основі кількості рядків коду.

Управління конфігурацією програмної системи

Конфігураційне управління – це **процес контролю змін у програмному забезпеченні**, що дозволяє підтримувати його стабільність та узгодженість.

Основні елементи управління конфігурацією

1. **Ідентифікація версій компонентів** – контроль змін у коді.
2. **Контроль змін** – реєстрація та перевірка всіх змін.
3. **Облік статусу конфігурації** – документування змін.
4. **Аудит конфігурації** – перевірка відповідності стандартам.

Інструменти управління конфігурацією

1. **Git, SVN** – системи контролю версій.
2. **Docker, Kubernetes** – контейнеризація компонентів.
3. **Jenkins, GitLab CI/CD** – автоматизація процесу розгортання.

Приклад: Використання Git для управління версіями коду дозволяє командам працювати паралельно та відстежувати зміни.

Завдання 1. Самостійна робота.

1 Написати короткий опис життєвого циклу будь-якого програмного продукту, який ви використовуєте щодня.

2 Підготувати приклад, де порушення принципів ІПЗ призвело до проблем у розробці.

Оформити завдання у вигляді есе (2 стор) та зробити презентацію.

Завдання 2. Реферат.

Реферат повинен містити:

1. **Титульний аркуш**: Назва теми, ПІБ студента, навчальна група, рік.
2. **Вступ (1–2 сторінки)**: Короткий опис обраної теми, її актуальність.
3. **Основна частина (10–12 сторінок)**: Детальний аналіз теми з викладенням ключових концепцій, методів, моделей.
4. **Висновки (1 сторінка)**: Підсумки, практичні висновки та можливі рекомендації.
5. **Список використаних джерел**: Оформлення відповідно до стандартів (ДСТУ чи APA).

Теми рефератів:

1. Життєвий цикл програмного продукту та програмного процесу.
2. Огляд стандартів програмної інженерії. SWEBOK V.3

3. Методи програмної інженерії. Евристичні методи. Методи неформалізованих підходів.

4. Методи програмної інженерії. Формальні методи. Методи прототипування.

5. Основні концепції, методи та технології сучасной програмной інженерії.

6. Інструменти (CASE) програмної інженерії. Інструменти забезпечення якості (Software Quality Tools).

7. Інструменти супровождення (Software Maintenance Tools). Інструменти конфігураційного управління (Software Configuration Management Tools)

8. Інструменти управління інженерної діяльністю (Software Engineering Management Tools). Інструменти підтримки процесів (Software Engineering Process Tools)

9. Планування та відстеження проєкту програмного забезпечення .

10. Когнитивні моделі складних систем

11. Реінженерія програмних систем: методи та практики.

12. Методи верифікації та формального доведення коректності програм.

13. Інженерія додатків та її зв’язок із предметною областю.

14. Методи проектування архітектури ПЗ: монолітна vs мікросервісна архітектура.

15. Патерни проєктування в програмній інженерії.

16. Методи рефакторингу програмного коду.

Навчальне видання

Методичні рекомендації
з дисципліни «Методологія інженерії програмного забезпечення»
для студентів спеціальності 121 «Інженерія програмного забезпечення»

Укладачі:
Чередніченко Ольга Юріївна
Шматко Олександр Віталійович
Літвінова Юлія Сергіївна
Сутягін Олександр Олександрович

Відповідальний за випуск доц. Копп А.М.
Роботу до видання рекомендував проф. Гамаюн І.П.

В авторській редакції

План 2025 р., поз. 173

Підп. до друку 13.02.2025.
Гарнітура Times New Roman.
Ум. друк. арк. 0,35.

Видавничий центр НТУ «ХПІ».
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, Харків, вул. Кирпичова, 2

Самостійне електронне видання