Гаврилюк: 1, 9, 17

Єфімець: 2, 10, 18

Кібукевич: 3, 11, 19

Курач: 4, 12, 20

Матвійчук: 5, 13, 21

Назар: 6, 14, 22

Рибачек: 7, 15, 23

Свиридюк: 8, 16, 24

**1. Інженерні основи програмного забезпечення/основні визначення програмної інженерії**

**Програмна інженерія** — це застосування [системного](https://uk.m.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D0%B8%D0%B9_%D0%BF%D1%96%D0%B4%D1%85%D1%96%D0%B4), вимірного підходу до [розроблення](https://uk.m.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%B5_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F), використання та [супроводу](https://uk.m.wikipedia.org/wiki/%D0%A1%D1%83%D0%BF%D1%80%D0%BE%D0%B2%D1%96%D0%B4_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F) програмного забезпечення, та дослідження цих підходів, тобто застосування принципів [інженерії](https://uk.m.wikipedia.org/wiki/%D0%86%D0%BD%D0%B6%D0%B5%D0%BD%D0%B5%D1%80%D1%96%D1%8F) до [програмного забезпечення](https://uk.m.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%B5_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F). Вперше термін був використаний в 1968 році на конференції з програмної інженерії, що була організована Науковим комітетом [NATO](https://uk.m.wikipedia.org/wiki/NATO).

Еталонну модель програмної інженерії можна визначити як взаємодію трьох факторів: процесів; продуктів; ресурсів.  
Кожна програмна система протягом свого існування проходить з певною послідовністю фази або стадії від задуму до його втілення в програми, експлуатацію та вилучення. Така послідовність фаз називається *життєвим циклом розробки (Software life cycle processes).* На кожній фазі відбувається певна сукупність процесів, кожен з яких породжує певний продукт, використовуючи певні ресурси.  
Усі продукти всіх процесів програмної інженерії являють собою певні описи — тексти вимог до розробки, погодження домовленостей, документацію, тексти програм, інструкції з експлуатації тощо.  
Головними ресурсами програмної інженерії, які визначають ефективність її розробок, є час і вартість цих розробок.  
Різновиди діяльності, котрі становлять процеси життєвого циклу програмної системи, зафіксовано в міжнародному стандарті ISO/IEC 12207 :Informational Technology - Software life cycle processes.  
Згідно з наведеним стандартом, усі процеси поділено на три групи:

головні процеси;

допоміжні процеси;

організаційні процеси.  
До головних процесів віднесено такі:

процес придбання, який ініціює життєвий цикл системи та визначає організацію-покупця автоматизованої системи, програмної системи або сервісу;

процес розроблення, який означає дії організації — розробника програмного продукту;

процес постачання, який означає дії під час передавання розробленого продукту покупцеві;

процес експлуатації, який означає дії з обслуговування системи під час її використання — консультації користувачів, вивчення їхніх побажань тощо;

процес супроводження, який означає дії з керування модифікаціями, підтримки актуального стану та функціональної придатності, інсталяцію та вилучення версій програмних систем у користувача.  
У свою чергу, до процесу розроблення входять такі процеси:

інженерія вимог до системи;

проектування;

кодування й тестування.  
До допоміжних процесів віднесено ті, що так чи інакше забезпечують якість продукту. Терміном *якість продукту* позначено сукупність властивостей, які зумовлюють можливість задоволення потреб замовника, котрий сформулював їх у формі вимог на розробку.  
До організаційних процесів віднесено менеджмент розробки, створення структури організації, навчання персоналу, визначення відповідальності кожного з учасників процесів життєвого циклу розробки.  
Стандарт, розглянутий нами вище, є головним чинником визначення змісту діяльності у сфері програмної інженерії, і всі знання, яких потребують професіонали з програмної інженерії, формулюються стосовно процесів, визначених стандартом ISO/IEC 12207:1995 - 0801: Informational Technology - Software life cycle processes.   
Зупинимося докладніше на процесах розробки програмного забезпечення, які в сукупності мають забезпечити шлях від усвідомлення потреб замовника до передавання йому готового продукту. На цьому шляху виділяють низку характерних робіт.  
*Визначення вимог.* Збір та аналіз вимог замовника виконавцем і подання їх у нотації, яка є зрозумілою як для замовника, так і для виконавця.  
*Проектування.* Перетворення вимог до розробки в послідовність проектних рішень щодо способів реалізації вимог: формування загальної архітектури програмної системи та принципів її прив’язки до конкретного середовища функціонування; визначення детального складу модулів кожної з архітектурних компонент.  
*Реалізація.* Перетворення проектних рішень на програмну систему, яка реалізує такі рішення.  
*Тестування.* Перевірка кожного з модулів та способів їхньої інтеграції; тестування програмного продукту в цілому (так звана верифікація); тестування відповідності функцій працюючої програмної системи *вимогам (requirements)*, поставленим до неї замовником (так звана валідація).  
Експлуатація та супроводження готової програмної системи.  
За десятиріччя досвіду з побудови програмних систем напрацьовано низку типових схем послідовності наведених вище робіт. Такі схеми назвали моделями життєвого циклу. Історично першою застосовувалася так звана *водоспадна* або *каскадна модель,* за якою вважалося, що кожна з робіт виконується один раз і в тому порядку, в якому їх перелічено вище. Інакше кажучи, робилося припущення, що кожну з робіт буде виконано настільки ретельно, що після її завершення й переходу до наступної роботи повернення до попередньої не потрібно. На рис. 2.1 показано послідовність робіт за *водоспадною (каскадною) моделлю.* Повернення до початкової стадії робіт передбачається, як бачимо, лише як результат супроводження.  
Цінність такої моделі полягає в тому, що вперше було зафіксовано послідовність процесів розроблення та стадії готовності програмного продукту, а недоліком є те, що в її концепцію покладено модель фабрики, коли продукт проходить стадії від задуму до виробництва, після чого передається замовникові як готовий виріб, зміну якого непередбачено, хоча й можлива заміна на інший подібний продукт у разі рекламації.

**2. Основи інженерії вимог до програмного забезпечення/ Характеристика областей знань з інженерії**

Вимоги до програмного забезпечення — набір вимог щодо властивостей, якості та функцій програмного забезпечення, що буде розроблено, або знаходиться у розробці. Вимоги визначаються в процесі аналізу вимог та фіксуються в специфікації вимог, діаграмах прецедентів та інших артефактах процесу аналізу та розробки вимог.

Розробка вимог до програмної системи може бути розділена на декілька етапів:

Знаходження вимог (збір, визначення потреб зацікавлених осіб та систем).

Аналіз вимог (перевірка цілісності та закінченості).

Специфікація (документування вимог).

Тестування вимог.

Різні методології розробки програмного забезпечення по-різному працювали з вимогами. В дуже старій, та не актуальній моделі водоспаду (англ. waterfall) етап аналізу та розробки вимог є першим. Особливістю є те, що він повністю закінчується до початку проектування та розробки ПЗ, а останні не можуть початися до завершення аналізу вимог.

Карл Вігерс визначає три рівні вимог до програмного забезпечення.

Бізнес-вимоги — визначають призначення ПЗ, можуть описуватися в документі про бачення (англ. vision) та документі про межі проекту (англ. scope).

Вимоги користувача — визначають набір завдань користувача, які повинна вирішувати програма, а також сценарії їхнього вирішення в системі. Ці вимоги можуть мати вигляд тверджень, варіантів використання, історій користувача, сценаріїв взаємодії.

Функціональні вимоги — визначають «що» повинен робити програмний продукт. Ці вимоги описуються в документі

В ітеративних процесах розробки фаза аналізу та розробки вимог в різному об'ємі є на кожній ітерації.

.Характеристика областей знань з інженерії програмного забезпечення – SWEBOK

У даному пункті розглядається теоретичний і інтелектуальний базис проектування – методи, принципи, засоби і методології, представлені областями в ядрі знань програмної інженерії SWEBOK.

Ядро знань SWEBOK – основний науково-технічний документ, що відображає знання та досвід багатьох іноземних і вітчизняних фахівців з програмної інженерії [1–10] і узгоджується з регламентованими процесами ЖЦ стандарту ISO/IEC 12207.

Документ містить у собі опис 10 областей, кожна з яких представлена відповідно до прийнятої всіма учасниками формування ядра SWEBOK загальної схеми опису, що містить у собі визначення понятійного апарату, методів і засобів, а також інструментів підтримки інженерної діяльності. Стосовно кожної області визначено коло знань, які повинні практично використовуватися при виконанні процесів життєвого циклу.

Для подання понятійного апарату областей знань SWEBOK проведемо умовне розділення областей на головні і допоміжні організаційні області.

У кожній області наведені ключові поняття, підходи і методи проектування різних типів ПС. Розподіл областей на основні і допоміжні відповідає структурі розподілу процесів стандарту ISO/IEC 12207, виконання яких визначається методами і засобами, запропонованими в ядрі знань SWEBOK.

Далі подається огляд кожної області цього ядра, визначається її роль у проектуванні і реалізації програмних продуктів. У деяких підрозділах показаний зв’язок з положеннями відповідних стандартів, що регламентують і регулюють виконання процесів проектування програмних систем.

**3. Ліцензування програмного продукту.**

Ліцензування програмного забезпечення - це процедура отримання розробником виняткових авторських та майнових прав на використання розробленого ним продукту. Покупцю цього продукту надається право на легальне використання придбаних ним копій.

**4. Аналіз вимог замовника до програмного продукту. Розробка технічного завдання.**

Аналіз вимог замовника до програмного продукту є ключовим етапом у процесі розробки програмного забезпечення. Від того, наскільки точно і повно будуть зрозумілі вимоги замовника, залежить успіх проекту. Давайте розглянемо цей процес більш детально.

#### 1. Визначення вимог

1.1. Ідентифікація зацікавлених сторін

* Замовник: Організація або особа, яка фінансує проект.
* Користувачі: Ті, хто буде безпосередньо використовувати систему.
* Команда розробників: Включає проект-менеджера, програмістів, тестувальників, дизайнерів тощо.

1.2. Методи збору вимог

* Інтерв'ю: Персональні бесіди з ключовими зацікавленими сторонами для збору детальної інформації.
* Анкетування: Використання опитувальників для збору даних від великої групи користувачів.
* Наради та воркшопи: Колективні обговорення для виявлення та узгодження вимог.
* Аналіз документації: Вивчення існуючої документації, пов'язаної з поточною системою або процесами.
* Спостереження: Пряме спостереження за користувачами в процесі їхньої роботи.

1.3. Вимоги високого рівня

* Бізнес-вимоги: Описують цілі та задачі, які повинна вирішувати система з точки зору бізнесу.
* Користувацькі вимоги: Описують задачі, які користувачі повинні мати можливість виконувати за допомогою системи.
* Системні вимоги: Визначають, як система повинна функціонувати для задоволення бізнес-вимог та користувацьких вимог.

#### 2. Аналіз вимог

2.1. Класифікація вимог

* Функціональні вимоги: Визначають конкретні функції та можливості системи (наприклад, можливість створення звітів, обробка замовлень).
* Нефункціональні вимоги: Визначають якісні характеристики системи (наприклад, продуктивність, безпека, зручність використання).

2.2. Верифікація та валідація вимог

* Верифікація: Перевірка того, що вимоги чітко сформульовані та зрозумілі.
* Валідація: Переконання в тому, що вимоги відповідають потребам замовника і користувачів.

2.3. Пріоритизація вимог

* Обов'язкові вимоги: Вимоги, які повинні бути реалізовані у будь-якому випадку.
* Бажані вимоги: Вимоги, які бажано реалізувати, якщо це можливо.
* Можливі вимоги: Вимоги, які можна реалізувати у випадку наявності додаткового часу та ресурсів.

#### 3. Документування вимог

3.1. Специфікація вимог до програмного забезпечення (SRS)

* Зміст специфікації: Повинен включати всі зібрані та проаналізовані вимоги.
* Структура SRS:
  + Вступ: Мета створення документа, загальний опис проекту.
  + Загальні вимоги: Описує загальні бізнес-цілі та обмеження.
  + Функціональні вимоги: Детальний опис функціональних можливостей системи.
  + Нефункціональні вимоги: Визначення нефункціональних характеристик системи.
  + Системні вимоги: Вимоги до апаратного та програмного забезпечення.

3.2. Використання діаграм

* Use Case діаграми: Описують сценарії використання системи.
* Діаграми активності: Візуалізують процеси та робочі потоки.
* Діаграми послідовностей: Показують взаємодію між користувачами та системою.

#### 4. Узгодження та затвердження вимог

4.1. Узгодження з зацікавленими сторонами

* Розгляд вимог: Спільне обговорення з замовником та користувачами.
* Внесення коректив: Коригування вимог відповідно до отриманих зауважень та пропозицій.

4.2. Затвердження вимог

* Формальне затвердження: Підписання документа специфікації вимог замовником.
* Остаточне закріплення: Всі зміни у вимогах після затвердження повинні проходити через формальний процес зміни вимог.

### Розробка технічного завдання

Технічне завдання (ТЗ) – це офіційний документ, який містить детальні вимоги до розробки програмного продукту. Цей документ служить дорожньою картою для всієї команди розробників, визначаючи всі аспекти проекту, від функціональних вимог до технічних обмежень і критеріїв приймання.

#### Основні компоненти технічного завдання

1. Вступ
   * Мета документа: Описує ціль створення ТЗ, для чого цей документ призначений.
   * Область застосування: Вказує, де і як буде використовуватися система, цільову аудиторію.
   * Визначення термінів і скорочень: Забезпечує розуміння специфічних термінів і скорочень, використаних у ТЗ.
2. Загальні положення
   * Цілі та завдання проекту: Описи основних цілей, які повинна досягнути система.
   * Принципи побудови системи: Вказує основні принципи, на яких базується архітектура системи.
3. Функціональні вимоги
   * Опис функціональності: Докладний перелік функцій, які повинна виконувати система.
   * Вхідні та вихідні дані: Описує, які дані вводяться в систему і які дані вона повинна генерувати.
   * Бізнес-логіка та алгоритми: Визначає основні алгоритми та правила бізнес-логіки.
4. Нефункціональні вимоги
   * Продуктивність: Вимоги до швидкодії та обробки даних.
   * Безпека: Вимоги до захисту даних та доступу до системи.
   * Зручність використання (Usability): Вимоги до інтерфейсу користувача та його зручності.
   * Надійність: Вимоги до безвідмовної роботи системи.
   * Масштабованість: Вимоги до можливості розширення системи.
5. Вимоги до інтерфейсу користувача
   * Опис інтерфейсів: Описує основні екрани та діалогові вікна системи.
   * Вимоги до дизайну: Вказує вимоги до зовнішнього вигляду інтерфейсу.
6. Системні вимоги
   * Апаратне забезпечення: Визначає мінімальні та рекомендовані вимоги до апаратного забезпечення.
   * Програмне забезпечення: Визначає вимоги до програмного забезпечення, яке буде використовуватися.
7. Обмеження та припущення
   * Технічні обмеження: Визначає технічні обмеження, які можуть впливати на розробку системи.
   * Припущення: Вказує припущення, на яких базується розробка системи.
8. Критерії приймання
   * Методики тестування: Описує методи та критерії, які будуть використовуватися для перевірки відповідності системи вимогам.
   * Умови приймання: Визначає умови, при яких система буде вважатися прийнятою.

#### Процес розробки технічного завдання

1. Збір вимог
   * Використання інтерв'ю, анкет, нарад та аналізу документації для збору вимог від замовника та користувачів.
2. Аналіз вимог
   * Класифікація вимог на функціональні та нефункціональні, їх пріоритизація та перевірка на відповідність цілям проекту.
3. Документування вимог
   * Створення детального документа, що містить усі вимоги до системи, з описами, діаграмами та специфікаціями.
4. Узгодження з зацікавленими сторонами
   * Обговорення та внесення коректив у документ разом із замовником та користувачами для забезпечення повного розуміння та узгодженості.
5. Затвердження технічного завдання
   * Формальне затвердження документа замовником, після чого він стає основою для подальшої розробки та тестування системи.

**5. Основи моделювання програмного забезпечення**

Моделювання програмного забезпечення – це процес створення абстрактного представлення системи або її частин, що допомагає зрозуміти, розробити і втілити програмне забезпечення. Ось основні концепції моделювання програмного забезпечення

1) Мета моделювання:

* Зрозуміння системи: Моделі допомагають розробникам, клієнтам і іншим зацікавленим сторонам зрозуміти структуру та поведінку системи.
* Аналіз і проектування: Моделювання дозволяє виявити потенційні проблеми на ранніх етапах і спроектувати систему, що відповідає вимогам.

Типи моделей

* Статичні моделі: Представляють структуру системи, зокрема класові діаграми та діаграми об'єктів.
* Динамічні моделі: Описують поведінку системи в часі, зокрема діаграми активності, послідовності та станів.

Уніфікована мова моделювання (UML)

* **UML**: Це стандартна мова для моделювання програмних систем. UML включає в себе різні типи діаграм для представлення різних аспектів системи.

**6. Створення діаграми класів.**

Діаграма класів є одним з найважливіших інструментів для візуалізації та проектування об'єктно-орієнтованих систем. Вона відображає класи системи, їх атрибути, методи та взаємозв'язки між ними. Створення діаграми класів є критично важливим етапом в процесі проектування програмного забезпечення, оскільки вона допомагає розробникам зрозуміти структуру системи, виявити потенційні проблеми та спростити комунікацію в команді.

Щоб створити діаграму класів, необхідно виконати наступні кроки:

1. Ідентифікація класів: Перш за все, необхідно визначити всі класи, які будуть використовуватися в системі. Класи можна ідентифікувати, аналізуючи вимоги до системи та виділяючи сутності, які мають бути представлені як класи.
2. Визначення атрибутів та методів: Для кожного класу необхідно визначити відповідні атрибути (поля даних) та методи (функції, які оперують даними). Атрибути представляють стан об'єкта, тоді як методи визначають його поведінку.
3. Візуалізація класів: На діаграмі класи зображуються у вигляді прямокутників, розділених на три секції: назва класу, атрибути та методи. Назва класу розміщується у верхній секції, атрибути - у середній, а методи - у нижній.
4. Зв'язки між класами: Після візуалізації класів необхідно встановити зв'язки між ними. Існують різні типи зв'язків, такі як асоціація, композиція, агрегація та успадкування. Ці зв'язки зображуються за допомогою різних типів ліній та стрілок.
5. Опис зв'язків: Для кожного зв'язку необхідно вказати його тип, напрямок (односторонній чи двосторонній) та, за необхідності, роль або множинність.
6. Додаткові елементи: Залежно від потреб, на діаграмі класів можна додати додаткові елементи, такі як інтерфейси, пакети або примітки.

Створення діаграми класів зазвичай виконується за допомогою спеціалізованих інструментів моделювання, таких як UML-редактори або діаграмні інструменти в середовищах розробки (IDE). Ці інструменти забезпечують зручний інтерфейс для створення, редагування та форматування діаграм класів.

Діаграма класів є потужним інструментом для візуалізації структури об'єктно-орієнтованої системи та полегшення комунікації між розробниками. Вона допомагає зрозуміти взаємозв'язки між класами, виявити потенційні проблеми та спростити процес проектування та розробки програмного забезпечення.

**7. Створення діаграми взаємодії.**

Діаграма взаємодії (або діаграма послідовності) — це тип діаграми UML, яка показує, як об'єкти взаємодіють у межах певного сценарію використання. Вона зображає обмін повідомленнями між об'єктами в хронологічному порядку.

#### **Етапи створення діаграми взаємодії:**

1. **Визначення учасників (акторів і об'єктів)**:
   * Визначте всі об'єкти та актори, які братимуть участь у взаємодії. Наприклад, користувач, система, база даних, зовнішні сервіси тощо.
2. **Визначення сценарію взаємодії**:
   * Опишіть послідовність кроків або дій, які виконують об'єкти та актори. Це допоможе побудувати діаграму взаємодії.
3. **Побудова діаграми**:
   * Використовуйте спеціалізовані інструменти або нотації для побудови діаграми. Існують різні інструменти для створення діаграм UML, такі як Lucidchart, draw.io, Visio або навіть текстові представлення, такі як Mermaid.

### **Приклад створення діаграми взаємодії**

#### **Сценарій: Процес здачі екзамену онлайн**

1. **Учасники**:
   * Студент
   * Система управління навчанням (LMS)
   * Екзаменатор
   * Система перевірки відповідей
2. **Сценарій взаємодії**:
   * Студент входить у систему.
   * Студент вибирає екзамен.
   * Студент проходить екзамен.
   * LMS передає відповіді на перевірку.
   * Система перевірки відповідей обробляє результати.
   * Результати екзамену повертаються студенту та екзаменатору.

#### **Діаграма взаємодії (Mermaid)**

sequenceDiagram

participant Студент

participant LMS

participant Екзаменатор

participant Система\_перевірки

Студент->>LMS: Вхід в систему

Студент->>LMS: Вибір екзамену

Студент->>LMS: Проходження екзамену

LMS->>Система\_перевірки: Передача відповідей на перевірку

Система\_перевірки->>LMS: Результати перевірки

LMS->>Студент: Відображення результатів

LMS->>Екзаменатор: Надсилання результатів екзаменатору

### **Інструменти для створення діаграм взаємодії:**

1. **Mermaid**:
   * Відмінний інструмент для створення діаграм за допомогою текстового опису.
   * Синтаксис простий і зрозумілий.
2. **Lucidchart**:
   * Онлайн-інструмент для створення діаграм з підтримкою UML.
   * Має інтуїтивний інтерфейс і багато функцій для командної роботи.
3. **draw.io (Diagrams.net)**:
   * Безкоштовний інструмент для створення різних типів діаграм, включаючи UML діаграми.
   * Підтримує інтеграцію з Google Drive та іншими хмарними сервісами.
4. **Microsoft Visio**:
   * Потужний інструмент для створення діаграм з підтримкою UML.
   * Підходить для створення складних діаграм і схем.
5. **PlantUML**:
   * Інструмент для створення UML діаграм за допомогою текстового опису.
   * Інтегрується з багатьма IDE та іншими інструментами розробки.

Діаграми взаємодії допомагають візуалізувати послідовність обміну повідомленнями між об'єктами в межах певного сценарію. Це корисний інструмент для розробників, аналітиків та інших зацікавлених сторін для розуміння та документування процесів у системі. Використання відповідних інструментів для створення діаграм допомагає зробити цей процес більш ефективним та зрозумілим.

**8. Створення діаграми станів.**

Створення діаграми станів є важливим етапом моделювання поведінки об'єктів у системі. Діаграми станів (також відомі як діаграми станів переходів або Statechart diagrams) відображають можливі стани об'єкта та переходи між цими станами. Ось кроки, які допоможуть вам створити діаграму станів:

1. **Визначте об'єкт або систему**: Перш за все, виберіть об'єкт або систему, для якого будете створювати діаграму станів. Наприклад, це може бути автомат продажу, користувач програми, замовлення в системі управління замовленнями і т.д.
2. **Визначте стани об'єкта**: Перерахуйте всі можливі стани, у яких може перебувати об'єкт. Наприклад, для автомату продажу це можуть бути такі стани, як "Очікування монети", "Очікування вибору продукту", "Видача продукту" і т.д.
3. **Визначте події та переходи**: Визначте, які події спричиняють зміну станів об'єкта. Наприклад, подія "вставити монету" може перевести автомат з стану "Очікування монети" у стан "Очікування вибору продукту".
4. **Побудуйте діаграму**:
   * **Початковий стан**: Позначте початковий стан за допомогою заповненої круглої точки.
   * **Станові вузли**: Позначте кожен стан об'єкта прямокутником з заокругленими кутами.
   * **Переходи**: Зобразіть переходи між станами за допомогою стрілок. Кожну стрілку підпишіть подією, яка спричиняє перехід.
   * **Кінцевий стан**: Позначте кінцевий стан подвійним кругом.
5. **Перевірте логіку**: Перевірте, чи всі стани та переходи логічно зв'язані та чи немає пропущених подій або нелогічних станів.

**9. Створення діаграми активностей.**

Діаграма діяльності — це блок-схема, яка показує, як одна діяльність призводить до іншої. Цю дію можна назвати системною операцією. Одна операція веде до наступної в потоці керування. Цей потік може бути паралельним, одночасним або розгалуженим. Діаграми діяльності використовують багато функцій, таких як fork, join тощо, щоб справлятися з усіма типами керування потоком. Подібно до інших діаграм, діаграми діяльності служать схожим фундаментальним цілям. Він фіксує динамічну поведінку системи.Діаграми діяльності будують виконувану систему, використовуючи підходи прямого та зворотного проектування. Це також візуалізація динамічної природи системи. Частина повідомлення – це єдиний елемент, якого не вистачає на діаграмі активності. Потік повідомлень від однієї дії до іншої не відображається. Іноді замість блок-схеми використовується діаграма діяльності. Діаграми не є блок-схемами, незважаючи на їх зовнішній вигляд. Він відображає різні потоки, включаючи одинарні, паралельні, розгалужені та одночасні.**Символ початку**

Він представляє на діаграмі діяльності початок процесу або робочого циклу. Його можна використовувати окремо або разом із символом примітки, який описує початкову позицію.

**Символ рішення**

Показано рішення та принаймні два шляхи розгалужуються з мовою умов, щоб користувачі могли бачити свої можливості. Символ служить рамкою або контейнером для ілюстрації розгалуження або з’єднання численних потоків.

**Примітка Символ**

Дозволяє тим, хто бере участь у створенні діаграми або в співпраці, передавати додаткові повідомлення, які не належать до діаграми. Додайте зауваження, щоб підвищити ясність і конкретність.

**Роз'єм символ**

Він відображає спрямований потік діяльності або потік керування. Вхідна стрілка ініціює крок дії; після завершення кроку потік переходить до вихідної стрілки.

**Об'єднаний символ/панель синхронізації**

Він поєднує два поточні процеси та повторно вводить їх у потік, у якому одночасно активний лише один процес. Для його зображення використовується товста вертикальна або горизонтальна лінія.

**Вилка символ**

Він розділяє один потік діяльності на два паралельні процеси. Вона зображується у вигляді з'єднання кількох стрілчастих ліній.

**Діяльність символ**

Показує дії, які складають змодельований процес. Основними компонентами діаграми діяльності є ці символи, кожен з яких має короткий опис, включений у його дизайн.

**Символ кінця**

Він відображає завершення всіх потоків діяльності та кінець діяльності.

Перевагами такої діаграми є:

◆ Ця діаграма може представляти умовні або одночасні процеси. Діаграми діяльності в інформаційних технологіях описують реальну поведінку робочого процесу системи. Ця діаграма описує фактичний стан діяльності системи, зображуючи всю послідовність виконаних дій.

◆ Аналітики та зацікавлені сторони зазвичай можуть легко зрозуміти діаграму діяльності.

◆ Він легкий для розуміння як BA, так і кінцевими користувачами. Діаграма активності в UML для ІТ-бізнес-аналітика – це та, яку IT-BA вважає найбільш корисною для ілюстрації робочого процесу.

◆ Вони зазвичай розглядаються як найважливіший інструмент у наборі інструментів аналітика, оскільки вони є одними з найбільш зручних для користувача діаграм.

**10. Створення діаграми пакетів.**

Діаграмою пакетів є діаграма, що містить пакети класів і залежності між ними. Строго кажучи, пакети та залежності є елементами діаграми класів, тобто діаграма пакетів – це всього лише форма діаграми класів. Однак на практиці причини побудови таких діаграм різні. Залежність між двома елементами має місце в тому випадку, якщо зміни у визначенні одного елемента можуть спричинити за собою зміну в іншому. Що стосується класів, то причини залежностей можуть бути дуже різними: один клас посилає повідомлення іншому; один клас включає частину даних іншого класу; один клас посилається на інший як на параметр операції. Якщо клас змінює свій інтерфейс, то будь-яке повідомлення, яке він посилає, може стати неправильним.

В ідеальному випадку тільки зміни в інтерфейсі класу повинні впливати на інші класи. Мистецтво проектування великих систем включає в себе мінімізацію залежностей, яка знижує вплив змін і вимагає менше зусиль на їх внесення.

Пакети є життєво необхідним засобом для великих проектів. Їх слід використовувати в тих випадках, коли діаграма класів, що охоплює всю систему в цілому і розміщена на єдиному аркуші паперу формату А4, стає складною для розуміння.

Пакети не дають відповіді на питання, яким чином можна зменшити кількість залежностей в системі, що розробляється, проте вони допомагають виділити ці залежності. Зведення до мінімуму кількості залежностей дозволяє знизити зв'язаність компонентів системи. Але евристичний підхід до цього процесу далекий від ідеалу.

Пакети особливо корисні при тестуванні. Кожен пакет при тестуванні може містити один або кілька тестових класів, за допомогою яких перевіряється поведінка пакета.

**11. Структура та архітектура програмного забезпечення.**

Архітектура програмного забезпечення (англ. software architecture) - це структура програми або обчислювальної системи, яка включає програмні компоненти, видимі зовні властивості цих компонентів, а також відносини між ними. Цей термін стосується також документування архітектури програмного забезпечення.

Структура програмного забезпечення

Персональний комп’ютер здатний виконувати будь-які дії з обробки інформації. Проте для цього необхідно скласти для комп’ютера на зрозумілій йому мові точну та докладну послідовну інструкцію (тобто програму), що показує, як саме треба обробляти інформацію. Сам по собі комп’ютер не має знань у жодній галузі свого використання, а всі ці знання зосереджені у програмах, що виконуються на комп’ютері.

Змінюючи програми для комп’ютера, можна перетворити його в робоче місце бухгалтера або конструктора, економіста або агронома, редагувати на ньому документи або грати в будь-яку гру. Тому для ефективного використання комп’ютера обов’язково потрібно знати про призначення та властивості програм, необхідних для роботи з ним.

Програми, які використовуються для роботи з комп’ютером, можна поділити на три категорії:

* системні програми, які виконують різноманітні допоміжні функції, наприклад, використовуються для створення копій наявної інформації, для перевірки роботоздатності пристроїв комп’ютера і т. ін.;
* прикладні програми, що безпосередньо забезпечують виконання необхідних користувачу робіт: редагування текстів, створення малюнків, обробка інформаційних масивів тощо;
* інструментальні системи (системи програмування), які забезпечують створення нових програм для комп’ютера.

Програмне забезпечення має ієрархічну структуру, схема якої показана нижче.

Схема ієрархічної структури програмного забезпечення

******

**12.Технології розробки програмного забезпечення**

Розробка програмного забезпечення (ПЗ) є складним і багатогранним процесом, що включає в себе використання різноманітних технологій. Ці технології допомагають розробникам створювати ефективні, надійні та зручні у використанні продукти. У цій статті ми розглянемо основні мови програмування, фреймворки, системи контролю версій, інструменти для безперервної інтеграції та доставки, контейнеризацію, хмарні технології, бази даних, тестування та розробку інтерфейсів користувача.

#### Мови програмування

Мови програмування є фундаментом будь-якого ПЗ. Високорівневі мови, такі як Python, Java, C# і JavaScript, забезпечують швидку розробку та зручність для програмістів. Python популярний у сфері веб-розробки, автоматизації, аналізу даних і машинного навчання. Java та C# використовуються для створення корпоративних додатків та мобільних додатків, а JavaScript є основною мовою для веб-розробки.

Середньорівневі мови, такі як C++ і Rust, забезпечують баланс між продуктивністю та зручністю. Вони часто використовуються для системного програмування, розробки ігор та додатків, що вимагають високої продуктивності. Низькорівневі мови, такі як C і Assembly, дозволяють розробляти системне ПЗ, операційні системи та драйвери, де важлива максимальна продуктивність і контроль над апаратними ресурсами.

#### Фреймворки та бібліотеки

Фреймворки та бібліотеки допомагають прискорити процес розробки, надаючи готові рішення для поширених завдань. У веб-розробці фронтенд-фреймворки, такі як React, Angular і Vue.js, спрощують створення динамічних інтерфейсів користувача. Для бекенду популярні фреймворки Django (Python), Flask (Python), Spring (Java), Express (Node.js) та ASP.NET (C#).

У сфері мобільної розробки Android додатки зазвичай створюються за допомогою Kotlin та Java, тоді як iOS додатки розробляються на Swift та Objective-C. Кросплатформені фреймворки, такі як Flutter (Dart), React Native (JavaScript) та Xamarin (C#), дозволяють створювати додатки для різних платформ з однією кодовою базою. Для розробки ігор використовуються Unity (C#) та Unreal Engine (C++), які забезпечують потужні інструменти для створення високоякісних ігрових продуктів.

#### Системи контролю версій

Системи контролю версій є невід'ємною частиною розробки ПЗ, оскільки вони дозволяють відстежувати зміни у вихідному коді та координувати роботу команд. Git є найпопулярнішою системою контролю версій, а платформи GitHub, GitLab і Bitbucket надають зручні інструменти для спільної роботи. Subversion (SVN) також використовується в деяких корпоративних середовищах.

#### CI/CD інструменти

Інструменти для безперервної інтеграції та доставки (CI/CD) автоматизують процес збірки, тестування та розгортання ПЗ, що дозволяє швидше випускати нові версії продукту. Jenkins, GitLab CI/CD, CircleCI та Travis CI є популярними інструментами для налаштування процесів CI/CD, які допомагають забезпечити якість та стабільність ПЗ.

#### Контейнеризація та оркестрація

Контейнеризація дозволяє ізолювати додатки та їх залежності у контейнерах, що забезпечує портативність і спрощує розгортання. Docker є найпопулярнішим інструментом для контейнеризації. Kubernetes використовується для автоматизації розгортання, масштабування та управління контейнеризованими додатками.

#### Хмарні технології

Хмарні платформи, такі як Amazon Web Services (AWS), Microsoft Azure та Google Cloud Platform (GCP), надають широкий спектр послуг для зберігання даних, обчислень, аналітики та машинного навчання. Використання хмарних технологій дозволяє швидко масштабувати ресурси та знижувати витрати на інфраструктуру.

#### Бази даних

Бази даних є критично важливими для зберігання та управління даними додатків. Реляційні бази даних, такі як MySQL та PostgreSQL, забезпечують надійне зберігання структурованих даних. NoSQL бази даних, такі як MongoDB та Cassandra, використовуються для зберігання великих обсягів нереляційних даних та забезпечують високу продуктивність і масштабованість.

#### Інструменти для тестування

Тестування є важливим етапом розробки ПЗ, що допомагає забезпечити якість продукту. Unit testing інструменти, такі як JUnit (Java), NUnit (C#) та pytest (Python), використовуються для тестування окремих модулів коду. Для інтеграційного тестування використовуються Selenium та Cypress, а для тестування продуктивності – JMeter та LoadRunner.

#### Розробка інтерфейсів користувача

Дизайн інтерфейсів користувача є важливою складовою розробки ПЗ, оскільки від цього залежить зручність використання продукту. Інструменти для дизайну, такі як Sketch, Figma та Adobe XD, дозволяють створювати макети інтерфейсів. Фронтенд розробка здійснюється за допомогою HTML, CSS та JavaScript, а також CSS фреймворків, таких як Bootstrap та Tailwind CSS.

**13. Етапи розробки клієнт-серверного застосунку.**

1. Аналіз вимог

* Збір вимог: Виявлення функціональних та нефункціональних вимог від замовника або кінцевих користувачів.
* Документування вимог: Створення технічної документації, що описує вимоги до системи.

2. Проектування архітектури

* Вибір архітектури: Визначення архітектурного підходу (наприклад, тришарова архітектура, мікросервіси тощо).
* Розробка архітектурної діаграми: Візуалізація компонентів системи, їх взаємодії та зв'язків.

3. Проектування клієнтської частини

* Інтерфейс користувача (UI/UX): Створення макетів, прототипів і дизайн-проектів для інтерфейсу користувача.
* Вибір технологій: Вибір мов програмування та фреймворків для розробки клієнтської частини (наприклад, React, Angular, Vue.js).

4. Проектування серверної частини

* Вибір технологій: Вибір мов програмування та фреймворків для розробки серверної частини (наприклад, Node.js, Django, Spring).
* Проектування бази даних: Розробка схеми бази даних, вибір СУБД (наприклад, MySQL, PostgreSQL, MongoDB).

5. Розробка API

* Визначення API: Створення специфікації API для забезпечення взаємодії між клієнтом та сервером (наприклад, RESTful, GraphQL).
* Документування API: Створення документації для API (наприклад, за допомогою Swagger або Postman).

6. Реалізація клієнтської частини

* Розробка UI: Створення користувацького інтерфейсу відповідно до дизайн-проекту.
* Інтеграція з API: Зв'язок клієнтської частини з серверною через API.

7. Реалізація серверної частини

* Розробка бекенду: Створення серверної логіки, обробка запитів від клієнта.
* Інтеграція з базою даних: Налаштування доступу до бази даних, реалізація CRUD-операцій (створення, читання, оновлення, видалення).

8. Тестування

* Модульне тестування: Перевірка окремих компонентів системи на рівні коду.
* Інтеграційне тестування: Перевірка взаємодії між різними компонентами системи.
* Системне тестування: Комплексне тестування всієї системи на відповідність вимогам.
* Користувацьке тестування: Тестування системи з точки зору кінцевого користувача.

9. Деплоймент

* Налаштування середовища: Вибір та налаштування середовища для розгортання (наприклад, хмарні сервіси, локальні сервери).
* Розгортання застосунку: Встановлення та запуск застосунку в обраному середовищі.

10. Моніторинг та підтримка

* Моніторинг: Постійний нагляд за роботою застосунку, виявлення і виправлення проблем.
* Підтримка: Здійснення оновлень, виправлення помилок, додавання нових функцій.

**14. Тестування та супровід програмних продуктів**

Тестування та супровід програмних продуктів є критично важливими етапами в циклі розробки програмного забезпечення. Ці процеси забезпечують належну якість, стабільність і відповідність вимогам кінцевого програмного продукту. Розглянемо детальніше ці етапи та обов'язки, які лежать на тестувальниках.

Тестування програмних продуктів:

1. Планування тестування: Перш ніж розпочати тестування, необхідно розробити план тестування. Цей план визначає обсяг тестування, методи тестування, необхідні ресурси, графік виконання та критерії прийняття.
2. Підготовка тестових даних та середовища: Тестувальники повинні підготувати тестові дані (вхідні дані, очікувані результати) та налаштувати відповідне тестове середовище, яке імітує реальні умови експлуатації програмного продукту.
3. Функціональне тестування: Це один з найважливіших етапів, під час якого перевіряється відповідність функціональності програмного продукту заявленим вимогам. Тестувальники виконують різні тестові сценарії, перевіряючи правильність роботи функцій, модулів та компонентів системи.
4. Тестування інтерфейсу користувача (UI): Перевіряється зручність використання, навігація, візуальне представлення даних та загальний користувацький досвід.
5. Тестування продуктивності та навантаження: Перевіряється здатність програмного продукту витримувати очікуване навантаження, швидкодія, використання ресурсів та стабільність під навантаженням.
6. Тестування безпеки: Перевіряється стійкість програмного продукту до різних загроз безпеки, таких як зловмисні атаки, несанкціонований доступ та вразливості.
7. Регресійне тестування: Після внесення змін або виправлень у програмний продукт проводиться регресійне тестування, щоб переконатися, що нові зміни не вплинули негативно на раніше працюючі функції.
8. Звітування про дефекти: Під час тестування виявлені дефекти ретельно документуються у звітах про дефекти, які передаються розробникам для виправлення.
9. Верифікація та валідація: Після виправлення дефектів тестувальники перевіряють, чи було дефект дійсно виправлено, і чи програмний продукт відповідає вимогам замовника.

Супровід програмних продуктів:

1. Моніторинг та технічна підтримка: Після випуску програмного продукту тестувальники відповідають за моніторинг його роботи в реальних умовах експлуатації та надання технічної підтримки користувачам.
2. Звітування про проблеми та дефекти: Тестувальники збирають звіти про проблеми та дефекти, виявлені користувачами, і передають їх розробникам для аналізу та виправлення.
3. Планування та впровадження оновлень: Коли розробники виправляють дефекти або додають нові функції, тестувальники планують і виконують тестування оновлень перед їх випуском.
4. Документація та навчання: Тестувальники також можуть брати участь у підготовці технічної документації та навчанні користувачів або технічної підтримки.

Ключова відповідальність тестувальників полягає в забезпеченні належної якості програмного продукту на всіх етапах його життєвого циклу. Вони працюють в тісній співпраці з розробниками, аналітиками вимог та іншими зацікавленими сторонами, щоб гарантувати, що програмний продукт відповідає очікуванням і вимогам замовника, а також забезпечити безперебійну роботу та задоволеність користувачів.

**15. Тестування програмного продукту. Написання тестів з використанням бібліотеки модульного тестування.**

Тестування програмного продукту є критично важливим етапом у циклі розробки програмного забезпечення. Воно включає перевірку та оцінку продукту, щоб забезпечити його відповідність вимогам та очікуванням користувачів, а також виявлення можливих дефектів.

### **Основні типи тестування програмного продукту:**

1. **Модульне тестування** - перевірка окремих модулів або компонентів коду. Використання бібліотек, таких як unittest або pytest для Python.
2. **Інтеграційне тестування** - перевірка взаємодії між окремими модулями або компонентами. Виявлення проблем, які можуть виникнути при об'єднанні модулів.
3. **Системне тестування** - перевірка всієї системи в цілому. Оцінка системи на відповідність функціональним і нефункціональним вимогам.
4. **Приймальне тестування** - перевірка, чи відповідає система вимогам і очікуванням користувачів. Зазвичай виконується кінцевими користувачами або замовниками.
5. **Розробницьке тестування (Test-Driven Development, TDD)** - процес, в якому тести пишуться перед написанням коду. Цикл: Написати тест -> Написати код -> Виконати тест -> Рефакторинг.

### **Використання unittest:**

### unittest — це стандартна бібліотека для тестування в Python. Вона дозволяє створювати тести для окремих функцій та класів.

#### **Приклад:**

**Припустимо, у нас є файл calculator.py з функціями додавання та віднімання:**

**# calculator.py**

**def add(a, b):**

**return a + b**

**def subtract(a, b):**

**return a - b**

**Тепер створимо файл test\_calculator.py для написання тестів за допомогою unittest:**

**# test\_calculator.py**

**import unittest**

**from calculator import add, subtract**

**class TestCalculator(unittest.TestCase):**

**def test\_add(self):**

**self.assertEqual(add(1, 2), 3)**

**self.assertEqual(add(-1, 1), 0)**

**self.assertEqual(add(-1, -1), -2)**

**def test\_subtract(self):**

**self.assertEqual(subtract(2, 1), 1)**

**self.assertEqual(subtract(2, 2), 0)**

**self.assertEqual(subtract(2, 3), -1)**

**if \_\_name\_\_ == '\_\_main\_\_':**

**unittest.main()**

#### **Для unittest, щоб запустити тести з використанням unittest, відкрийте термінал і виконайте команду:**

**python test\_calculator.py**

**Ми створюємо клас TestCalculator, який успадковується від unittest.TestCase.**

**Методи тестування починаються з test\_.**

**Використовуємо методи self.assertEqual, щоб перевірити результати.**

**Написання модульних тестів є ключовим етапом забезпечення якості програмного забезпечення. Використання бібліотеки unittest дозволяє автоматизувати цей процес, роблячи його більш ефективним та зручним. Вибір бібліотеки залежить від ваших потреб та уподобань: unittest є частиною стандартної бібліотеки Python і підходить для простих випадків,**

**16. Огляд методу покриття операторів тестування.**

Структурне тестування направлено на тестування структури системи або компонента. Цей вид тестування, як правило, відносять до тестування «білого» та «сірого» ящиків, оскільки ми перевіряємо, що відбувається всередині системи або додатка.

### Методи структурного тестування:

1. *Строкове покриття (Statement Coverage)* – перевірка застосування усіх операторів в програмі на використання (хоча б один раз).
2. *Покриття шляху (Path Coverage)* – метод тестування, призначений для задоволення критеріїв охоплення кожного логічного шляху через програму.
3. *Покриття рішення (Branch Coverage)* – перевіряє, чи має кожна умова розгалуження для програми дійсні чи хибні значення;
4. *Покриття умови (Condition Coverage)* – метод, схожий на Branch Coverage, основна відмінність полягає в перевірці стану покриття для умовних і неумовних гілок.

### *Переваги:*

* можливість виявити та видалити «зайвий» код;
* можливість виявлення потенційних помилок на ранній стадії;
* забезпечує більш ретельне тестування ПЗ;
* не потребує високих витрат людино-годин.

### *Недоліки:*

* вимагає знання коду та інструментів тестування.

**17. Тестування методами “білого ящика”. Метод покриття умов.**

Тестування методом білого ящика (також: прозорого, відкритого, скляного ящика; засноване на коді або структурне тестування) – метод тестування програмного забезпечення, який передбачає, що внутрішня структура/пристрій/реалізація системи відомі тестувальнику. Ми вибираємо вхідні значення, грунтуючись на знанні коду, який буде їх обробляти. Точно так само ми знаємо, яким повинен бути результат цієї обробки. Знання всіх особливостей тестованої програми та її реалізації – обов’язкові для цієї техніки. Тестування білого ящика – поглиблення у код системи, за межі її зовнішніх інтерфейсів.

Згідно ISTQB:

**тестування білого ящика – це:**

* Тестування, засноване на аналізі внутрішньої структури компонента або системи;
* Тест-дизайн, заснований на техніці білого ящика – процедура написання або вибору тест-кейсів на основі аналізу внутрішнього устрою системи або компонента.

**Переваги:**

* Тестування може проводитися на ранніх етапах: немає необхідності чекати створення користувальницького інтерфейсу;
* Можна провести більш ретельне тестування, з покриттям великої кількості шляхів виконання програми.

**Недоліки:**

– Для виконання тестування білого ящика необхідно велика кількість спеціальних знань;

– При використанні автоматизації тестування на цьому рівні, підтримка тестових скриптів може виявитися достатньо накладною, якщо програма часто змінюється.

**18. Огляд методу модульного тестування ПЗ**

Модульне тестування — це метод ізоляції та тестування окремих одиниць коду для визначення ефективності кожного компонента. Замість тестування програмного забезпечення цей метод розбиває його на менші частини, щоб переконатися в коректності окремих компонентів.

Модульне тестування дозволяє програмісту, коли він буде змінювати код (проводити рефакторинг) бути впевненим, що модуль працює вірно (це — регресивне тестування). Оскільки модульне тестування вимагає написання тестів для всіх функцій та методів у програмі, помилки швидко локалізуються та виправляються.

**19. Моделі якості та надійності програмних систем**

Моделі якості та надійності програмних систем допомагають розробникам і менеджерам оцінювати, управляти та покращувати характеристики програмного забезпечення, забезпечуючи його відповідність вимогам користувачів і стандартам.

### **Моделі якості програмного забезпечення**

1. **Модель McCall'a**:
   * **Фактори якості**: відображають користувальницькі потреби (наприклад, коректність, надійність, ефективність).
   * **Критерії якості**: конкретні умови, які визначають кожен фактор.
   * **Метрики якості**: кількісні показники для вимірювання критеріїв.
2. **Модель Boehm'a**:
   * Деталізує якість за допомогою ціннісно-орієнтованих аспектів.
   * Має три рівні: висока, середня та низька складність.
3. **ISO/IEC 25010** (колишня ISO/IEC 9126):
   * Включає характеристики: функціональність, надійність, зручність використання, ефективність, переносимість, супроводжуваність.
   * Кожна характеристика має підхарактеристики.
4. **Модель Dromey'a**:
   * Зосереджується на властивостях компонентів та їх впливі на загальну якість системи.

### **Моделі надійності програмного забезпечення**

1. **Модель базової надійності**:
   * Використовує історичні дані про збої для прогнозування майбутніх показників надійності.
2. **Модель Musa**:
   * Враховує кількість помилок та частоту їх виникнення.
   * Зосереджується на інтенсивності відмов та середньому часі між відмовами (MTBF).
3. **Модель Jelinski-Moranda**:
   * Передбачає зменшення ймовірності відмов у міру усунення помилок.
4. **Модель Goel-Okumoto**:
   * Використовує процеси Пуассона для опису часу між відмовами.
5. **Модель Littlewood-Verrall**:
   * Інтегрує байєсівський підхід для оцінки надійності.
   * Враховує невизначеність в оцінці кількості помилок та ймовірностей їх виявлення.

### **Інтеграція моделей у процес розробки**

* **Agile та DevOps**: Використання моделей якості та надійності в гнучких методологіях розробки допомагає інтегрувати оцінку та покращення якості на кожному етапі життєвого циклу ПЗ.
* **Автоматизоване тестування**: Допомагає швидко виявляти та усувати помилки, підвищуючи надійність.
* **Метрики та звіти**: Регулярне використання метрик для оцінки якості та надійності дозволяє своєчасно виявляти проблеми та вживати заходів для їх вирішення.

### **Висновок**

Моделі якості та надійності програмних систем є важливими інструментами для забезпечення відповідності програмного забезпечення вимогам користувачів, зниження кількості помилок та підвищення загальної надійності системи. Вони допомагають структурувати процеси оцінки та покращення ПЗ, забезпечуючи його високу якість та стабільну роботу.

**20. Методи керування програмним проектом.**

Управління програмними проектами є важливим аспектом розробки програмного забезпечення, який визначає успішність виконання проекту. Ефективне управління включає в себе координацію ресурсів, планування завдань, контроль прогресу та комунікацію з усіма зацікавленими сторонами. Існують різні методи управління проектами, які допомагають досягти цієї мети. Розглянемо деякі з них.

#### Традиційні методи управління проектами

Традиційні або класичні методи управління проектами, такі як Waterfall (Каскадна модель), передбачають послідовне виконання етапів розробки: від збору вимог до проектування, реалізації, тестування та впровадження. Цей підхід забезпечує чітку структуру та передбачуваність, що особливо важливо для великих і комплексних проектів з чітко визначеними вимогами. Втім, недоліком є його негнучкість – будь-які зміни на пізніх етапах можуть бути важкими та дорогими.

#### Гнучкі методи управління проектами

Гнучкі методи, або Agile підходи, акцентують увагу на гнучкості та адаптивності. Основною ідеєю Agile є ітеративна розробка з частими релізами продукту, що дозволяє швидко реагувати на зміни вимог і отримувати зворотний зв'язок від користувачів. Найпоширеніші методології Agile включають Scrum і Kanban.

Scrum передбачає поділ проекту на спринти – короткі цикли розробки (зазвичай 2-4 тижні), в кінці кожного з яких команда представляє робочий інкремент продукту. Scrum-майстер координує роботу команди, допомагаючи усунути перешкоди, а власник продукту визначає пріоритети завдань. Регулярні зустрічі, такі як щоденні стендапи та спринт-рітроспективи, сприяють покращенню процесів та взаємодії команди.

Kanban зосереджується на візуалізації робочого процесу та обмеженні незавершеної роботи. Дошка Kanban з картками, що представляють завдання, дозволяє команді бачити поточний стан проекту та визначати вузькі місця у процесі. Цей підхід добре підходить для проектів з постійно змінними вимогами та потребою у швидкому реагуванні на нові завдання.

#### Гібридні методи

Гібридні методи поєднують елементи традиційних та гнучких підходів, адаптуючи їх до конкретних потреб проекту. Наприклад, Scrumban поєднує структуру Scrum з гнучкістю Kanban, дозволяючи командам планувати роботу за допомогою спринтів, але водночас зберігати гнучкість у процесі виконання завдань.

#### Інструменти та технології

Сучасні інструменти управління проектами, такі як JIRA, Trello, Asana, Microsoft Project та інші, підтримують різні методології та полегшують планування, відстеження прогресу та комунікацію. Вони забезпечують централізоване місце для зберігання завдань, документів і звітів, що сприяє прозорості та ефективності управління **проектом.**

**21. Завантаження коду до репозиторію. Система контролю версій Git. Основи галуження у Git. Робота з гілками.**

* Система контролю версій Git є потужним інструментом для відстеження змін у коді та спільної роботи над проектами. Вона дозволяє вести історію змін, відновлювати попередні версії коду, працювати над різними версіями проекту та інтегрувати зміни від різних розробників.
* Для того, щоб завантажити код у репозиторій - в консоль гіта пишеться наступне:   
  git commit -m "[Повідомлення коміту]"
* Нові коміти не видаляють старі і таким чином, можна контролювати версії ПЗ, що розробляється і в разі чого відкотитися до стабільнішої або потрібної версії ПЗ. Це можна відтворювати завдяки декільком командам, такі як:

Відкат до потрібної версії: git checkout **hash** (hash - це хеш коміту, йомо можна дізнатися ввівши команду git log, яка виведе список комітів, а також: дату, хеш, імя коміту та опис (якщо він був доданий))

* Гілки в GitHub використовуються для управління різними версіями проекту, полегшення командної роботи та організації процесу розробки. Вони дозволяють розробникам працювати над новими функціями, виправленнями помилок або експериментальними змінами незалежно одна від одної.

Для створення гілки використовується команда:git branch <назва гілки>. Тобто для узагальнення: загальноприйнято різними розробниками, що гілка main/master - це основна гілка для стабільних версій ПЗ. Наступні створені гілки потрібні для контролю експериментальних версій ПЗ та інших потреб, щоб не засоряти гілку main/master.

**22. Розробка структури програмного проекту. Інтеграція проекту з системою управління версіями коду Git4**

1. Створення локального репозиторію Git

* Відкрити термінал або командний рядок.
* Створити нову порожню директорію для проекту за допомогою команди mkdir new\_project.
* Перейти до новоствореної директорії за допомогою команди cd new\_project.
* Ініціалізувати Git репозиторій командою git init.

1. Визначення структури проекту
   * Створити потрібні директорії та файли в корені проекту.
   * Типова структура проекту на Python може мати такий вигляд:
     + src - директорія з основним кодом програми.
     + tests - директорія з модульними тестами.
     + docs - директорія з документацією проекту.
     + examples - директорія з прикладами використання.
     + requirements.txt - файл з переліком залежностей проекту.
     + README.md - файл з описом проекту та інструкціями.
     + setup.py - файл для упаковки проекту.
     + .gitignore - файл з переліком файлів та директорій, які Git має ігнорувати.
2. Створення початкових файлів
   * У директорії src створити файл \_\_init\_\_.py для визначення Python пакету.
   * Додати початковий код програми у відповідні файли в директорії src.
   * У файлі setup.py додати конфігурацію для упаковки проекту.
   * У файлі README.md додати опис проекту та інструкції.
3. Створення першого коміту
   * Використати команду git add . для додавання всіх створених файлів до staging area.
   * Виконати git commit -m "Initial commit" для створення першого коміту в локальному репозиторії.
4. Створення віддаленого репозиторію
   * Зареєструватись на GitHub, GitLab чи іншому хостингу Git.
   * Створити новий віддалений репозиторій відповідно до інструкцій хостингу.
   * Скопіювати URL віддаленого репозиторію.
5. Підключення локального репозиторію до віддаленого
   * Виконати команду git remote add origin <remote\_repo\_url>, щоб додати віддалений репозиторій як "origin".
6. Вивантаження локального репозиторію на віддалений
   * Виконати git push -u origin master для вивантаження локального репозиторію на віддалений за замовчуванням.
7. Налаштування workflow для розробки
   * Визначити правила іменування гілок, наприклад, feature/\* для нових функцій, hotfix/\* для виправлення помилок тощо.
   * Для розробки нової функції створити гілку з master за допомогою команди git checkout -b feature/new-feature.
   * Регулярно комітити зміни в коді за допомогою команд git add та git commit.
   * Вивантажувати зміни на віддалений репозиторій командою git push origin feature/new-feature.
   * Після завершення роботи над функціоналом виконати git pull origin master для отримання останніх змін з віддаленого репозиторію.
   * Якщо виникнуть конфлікти злиття, вирішити їх вручну в редакторі коду.
   * Створити запит на злиття (pull request) з гілки feature/new-feature до master на хостингу Git.
   * Після перевірки коду та затвердження запиту на злиття виконати операцію злиття.
   * Видалити використану гілку функції локально та на віддаленому репозиторії.

Це забезпечить організовану структуру програмного проекту та його інтеграцію з системою управління версіями коду Git. Що дозволить відстежувати історію змін, працювати паралельно над різними функціями, легко вирішувати конфлікти та зберігати резервні копії проекту на віддаленому репозиторії.

**23. Документування програмних продуктів.**

Ключові аспекти документування програмних продуктів:

### **Типи документації**

1. **Користувацька документація**:
   * **Керівництва користувача**: Інструкції по використанню програмного продукту, пояснення функцій, поради та трюки.
   * **Онлайн-допомога**: Вбудована допомога або довідкові системи, доступні безпосередньо з програми.
   * **Часті питання (FAQ)**: Список найпоширеніших запитань користувачів з відповідями.
2. **Технічна документація**:
   * **Документація для розробників**: Пояснює структуру коду, архітектуру програмного забезпечення, API, використовувані технології та інструменти.
   * **Коментарі в коді**: Коментарі в коді допомагають зрозуміти, що робить кожен блок коду.
   * **Автоматизована документація**: Використання інструментів, таких як Sphinx для Python, для створення документації з коментарів в коді.
3. **Проектна документація**:
   * **Вимоги**: Опис вимог до програмного продукту, функціональних та нефункціональних вимог.
   * **Технічні завдання**: Детальний опис задач, які необхідно виконати для розробки програмного забезпечення.
   * **Дизайн документація**: Опис архітектури системи, діаграми UML, схеми баз даних тощо.

### **Інструменти для документування**

1. **Sphinx** - Інструмент для генерації документації з вихідного коду Python. Використовується для створення гарної документації у форматах HTML, PDF та інших.
2. **MkDocs** - Інструмент для створення статичної документації з файлів Markdown.
3. **Doxygen** - Популярний інструмент для автоматизованого створення документації з коментарів у коді для різних мов програмування, таких як C++, C, Python.
4. **Swagger/OpenAPI** - Використовується для документування RESTful API, надаючи інтерактивну документацію для веб-сервісів.
5. **Javadoc** - Інструмент для генерації API документації з коментарів в коді Java.

### **Приклади документування з використанням Sphinx**

#### **1. Інсталяція Sphinx.** Для інсталяції Sphinx, виконайте команду pip install sphinx

#### **2. Створення документаційного проекту.** Перейдіть у директорію вашого проекту і виконайте команду sphinx-quickstart Ця команда запустить майстер, який допоможе налаштувати ваш документаційний проект.

#### **3. Документування функцій.** Додайте коментарі до вашого коду у форматі reStructuredText (reST):

def add(a, b):

"""

Додає два числа.

:param a: Перше число.

:param b: Друге число.

:return: Сума чисел a і b.

"""

return a + b

**4. Генерація документації.** Після налаштування конфігураційного файлу (conf.py), ви можете згенерувати документацію, виконавши команду make html. Це створить HTML версію вашої документації.

### **Важливі аспекти документування**

1. **Актуальність**: Документація повинна бути актуальною і відповідати поточному стану програмного забезпечення.
2. **Зрозумілість**: Використовуйте чітку і зрозумілу мову, щоб документація була легкою для розуміння.
3. **Повнота**: Документація повинна охоплювати всі аспекти програмного продукту, включаючи всі функції та можливості.
4. **Структурованість**: Добре організована документація з чіткою структурою допомагає швидко знаходити потрібну інформацію.

Документування програмного продукту є критично важливим для забезпечення зрозумілості, підтримки та розвитку програмного забезпечення. Використання відповідних інструментів і методик допомагає створювати якісну документацію, яка буде корисною як для користувачів, так і для розробників.

**24. Розробка експлуатаційних документів.**

Експлуатаційні документи допомагають користувачам та операторам правильно та безпечно використовувати програмне чи апаратне забезпечення.

#### **Основні типи документів:**

1. **Керівництво користувача**: Інструкції з використання продукту.
2. **Інструкція з установки**: Процес інсталяції продукту.
3. **Керівництво адміністратора**: Налаштування та управління системою.
4. **Керівництво з технічного обслуговування**: Регулярне обслуговування та виправлення помилок.
5. **Інструкція з безпеки**: Заходи безпеки при використанні.

#### **Основні кроки:**

1. **Збір інформації**: Консультації з розробниками та користувачами.
2. **Аналіз аудиторії**: Врахування рівня підготовки користувачів.
3. **Планування та структура**: Створення плану документа.
4. **Написання**: Проста мова, використання візуальних матеріалів.
5. **Рецензування та тестування**: Перевірка точності та зрозумілості.
6. **Редагування та оформлення**: Коригування та професійне оформлення.
7. **Розповсюдження**: Доступність документів для користувачів.