**Питання до екзамену з дисципліни**

**«Основи програмної інженерії»»**

1. Ядро знань SWEBOK.

SWEBOK (Software Engineering Body of Knowledge) - це зібрання знань у галузі інженерії програмного забезпечення. Це офіційний стандарт, розроблений IEEE Computer Society, який описує ключові аспекти, принципи та практики у сфері розробки програмного забезпечення. SWEBOK є обширним довідником, який охоплює різні області знань, необхідні для успішної роботи у галузі інженерії програмного забезпечення.

SWEBOK включає в себе наступні області знань (Knowledge Areas - KA):

1. **Вимоги до програмного забезпечення:**
   * Розуміння та управління вимогами до програмного забезпечення.
2. **Проектування програмного забезпечення:**
   * Проектування структури програмного забезпечення та його компонентів.
3. **Конструювання програмного забезпечення:**
   * Процеси створення програмного коду та його тестування.
4. **Тестування програмного забезпечення:**
   * Методи та процеси тестування програм.
5. **Технічне обслуговування програмного забезпечення:**
   * Управління змінами в програмному забезпеченні після його впровадження.
6. **Управління конфігурацією програмного забезпечення:**
   * Управління версіями та змінами в програмному забезпеченні.
7. **Управління інженерією програмного забезпечення:**
   * Організаційні та управлінські аспекти в галузі інженерії програмного забезпечення.
8. **Процеси інженерії програмного забезпечення:**
   * Опис та управління процесами розробки програмного забезпечення.
9. **Моделі та методи інженерії програмного забезпечення:**
   * Застосування різних моделей та методів в інженерії програмного забезпечення.
10. **Якість програмного забезпечення:**
    * Оцінка та забезпечення якості програмного забезпечення.
11. **Професійна практика в інженерії програмного забезпечення:**
    * Етичні та професійні аспекти в інженерії програмного забезпечення.

SWEBOK служить основою для навчання та сертифікації інженерів з програмного забезпечення і надає загальний набір знань, який може бути використаний в індустрії для забезпечення якісної розробки програмного забезпечення.

2. Життєвий цикл програмного забезпечення.

Життєвий цикл програмного забезпечення (ЖЦ ПЗ) - це концепція, що описує всі етапи від розробки та створення програми до її впровадження, експлуатації, підтримки та відключення. ЖЦ ПЗ може бути представлений у вигляді ряду фаз, кожна з яких має свої характеристики та завдання. Основні фази життєвого циклу програмного забезпечення включають:

1. **Аналіз:**
   * Визначення потреб користувача, встановлення вимог до програмного забезпечення, аналіз проблем і визначення цілей проекту.
2. **Проектування:**
   * Розробка архітектури програмного забезпечення, визначення структури та інтерфейсів, вибір технологій і платформ.
3. **Реалізація (Конструювання):**
   * Написання програмного коду відповідно до визначених вимог та дизайну, виконання тестування коду.
4. **Тестування:**
   * Проведення різноманітних тестів для перевірки якості та відповідності вимогам, виявлення та виправлення помилок.
5. **Впровадження (Випуск):**
   * Розгортання програмного забезпечення на виробничому середовищі, навчання користувачів, забезпечення технічної підтримки.
6. **Експлуатація та супровід (Обслуговування):**
   * Забезпечення нормального функціонування програми в реальних умовах, виявлення та усунення можливих проблем, внесення змін у випадку необхідності.
7. **Вилучення (Відключення):**
   * Відключення програмного забезпечення, якщо воно застаріло чи більше не використовується.

ЖЦ ПЗ може бути представлений в різних варіаціях в залежності від конкретних методологій розробки (наприклад, водопадна, ітеративна, спільна розробка тощо) та особливостей проекту. Кожна фаза може включати в себе різні процеси та дії, спрямовані на забезпечення якості та ефективності розробки програмного забезпечення.

3. Основні стандарти пов‘язані з життєвим циклом програмного забезпечення.

Існує кілька стандартів, пов'язаних з життєвим циклом програмного забезпечення (ЖЦ ПЗ), які визначають рекомендації та норми для розробки, управління та підтримки програмного забезпечення. Основні стандарти в цій області включають:

1. **ISO/IEC 12207:**
   * Назва: Information technology – Software life cycle processes.
   * Опис: Цей стандарт визначає процеси життєвого циклу програмного забезпечення, включаючи вимоги, проектування, реалізацію, тестування, впровадження та підтримку.
2. **ISO/IEC 15288:**
   * Назва: Systems and software engineering – System life cycle processes.
   * Опис: Цей стандарт визначає загальні процеси життєвого циклу систем та програмного забезпечення та визначає їх взаємодію.
3. **ISO/IEC 20000-1:**
   * Назва: Information technology – Service management – Part 1: Service management system requirements.
   * Опис: Цей стандарт встановлює вимоги до систем управління послугами, включаючи управління життєвим циклом програмного забезпечення.
4. **IEEE 730:**
   * Назва: IEEE Standard for Software Quality Assurance Plans.
   * Опис: Цей стандарт визначає вимоги до плану забезпечення якості програмного забезпечення, включаючи визначення процесів, стандартів та практик, що використовуються для забезпечення якості.
5. **IEEE 1012:**
   * Назва: IEEE Standard for System and Software Verification and Validation.
   * Опис: Стандарт визначає вимоги до процесів верифікації та валідації в системному та програмному інженерії.
6. **ISO/IEC 25000:**
   * Назва: Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE.
   * Опис: Цей стандарт визначає концепції та вимоги до оцінки якості програмного забезпечення.
7. **ISO/IEC 29110:**
   * Назва: Systems and software engineering – Lifecycle profiles for Very Small Entities (VSEs).
   * Опис: Стандарт призначений для дуже малих підприємств і визначає профілі життєвого циклу, які відповідають їхнім конкретним потребам.

Ці стандарти допомагають організаціям розробляти, управляти та оцінювати програмне забезпечення згідно з визначеними нормами та вимогами, забезпечуючи високий рівень якості та ефективності в усьому життєвому циклі програмного забезпечення.Початок форми

4. Моделі життєвого циклу програми. Каскадна модель.

Каскадна модель (іноді називається "водоспадною") є однією з класичних моделей життєвого циклу програмного забезпечення. Ця модель передбачає лінійну та послідовну послідовність етапів розробки, які передаються від одного етапу до наступного, як вода, що стікає вниз по каскаду. Основні етапи каскадної моделі включають:

1. **Аналіз вимог:**
   * Збір і аналіз вимог до програмного забезпечення від замовника.
2. **Проектування:**
   * Визначення архітектури, структури та компонентів програми на основі вимог.
3. **Реалізація (Кодування):**
   * Написання програмного коду відповідно до визначеного дизайну.
4. **Тестування:**
   * Проведення різноманітних тестів для перевірки правильності та якості програми.
5. **Впровадження (Випуск):**
   * Впровадження програми в життя, встановлення на робочі станції або сервери.
6. **Обслуговування:**
   * Підтримка та виправлення помилок, які виявляються під час експлуатації програми.

Основні особливості каскадної моделі включають:

* **Лінійний характер:** Кожен етап починається тільки після завершення попереднього, і зміни на одному етапі можуть вимагати повернення до початку.
* **Жорстке визначення вимог:** Вимоги повинні бути повністю визначені на початку процесу, і будь-які зміни вимагають великих зусиль та ресурсів.
* **Приділення великої уваги документації:** Кожен етап супроводжується створенням документації, яка фіксує вимоги, дизайн, код, тестування тощо.
* **Можливість використання на початкових етапах проекту.**

Каскадна модель добре підходить для проектів з чітко визначеними вимогами та стабільним середовищем розробки. Однак вона може бути менш ефективною в умовах, коли вимоги нестабільні або важко передбачити заздалегідь. У відповідь на ці обмеження були розроблені інші моделі, такі як ітеративна, спіральна, агільна та інші.

5. Моделі життєвого циклу програми. Спіральна модель.

Спіральна модель життєвого циклу програмного забезпечення (ЖЦ ПЗ) є ітеративною моделлю, яка поєднує в собі елементи каскадної моделі з можливістю ітерацій і зворотного зв'язку. Ця модель була запропонована Баррі Боемом (Barry Boehm) і визначається спіральним обертанням, яке представляє собою послідовність етапів, що обертаються навколо центральної вісі. Кожен оберт моделі представляє одну ітерацію розробки.

Основні етапи спіральної моделі включають:

1. **Визначення об'єктів:**
   * Визначення цілей проекту, ідентифікація областей ризику, формування вихідних даних для ітерації.
2. **Оцінка ризиків:**
   * Оцінка ризиків та визначення стратегії керування ризиками.
3. **Розробка та тестування:**
   * Розробка, тестування та оцінка продукту в рамках обраної стратегії.
4. **Оцінка результатів ітерації:**
   * Оцінка результатів, прийняття рішення про продовження чи завершення проекту.

Кожна ітерація спіральної моделі проходить через чотири основні квадранти, що відображають різні аспекти проекту:

1. **Квадрант здійснення цілей (Objective Setting):**
   * Формулювання мети і цілей проекту.
2. **Квадрант ризиків (Risk Assessment):**
   * Визначення, оцінка і керування ризиками.
3. **Квадрант розробки та тестування (Development and Testing):**
   * Розробка, тестування та перевірка продукту.
4. **Квадрант оцінки результатів (Evaluation of Results):**
   * Оцінка результатів ітерації та вирішення, чи продовжити наступний оберт.

Основні особливості спіральної моделі включають:

* **Зорієнтована на ризики:** Акцент на визначенні та керуванні ризиками у кожній ітерації.
* **Гнучкість:** Можливість внесення змін і коригувань під час розробки.
* **Ітеративність:** Постійні ітерації дозволяють збагачувати та уточнювати продукт.
* **Підтримка зворотного зв'язку:** Забезпечення механізму зворотного зв'язку для вдосконалення процесу розробки.

Спіральна модель є ефективною для проектів, де важко визначити вимоги наперед, або коли ризики грають важливу роль у процесі розробки. Вона дозволяє адаптуватися до змін і вирішувати проблеми, які виникають під час реалізації проекту.

6. Моделі життєвого циклу програми. Еволюційна модель.

Еволюційна модель життєвого циклу програмного забезпечення є іншим варіантом ітеративної моделі, яка передбачає випуск невеликих частин продукту на кожній ітерації, поступове їх вдосконалення і розширення. Модель орієнтована на поступову еволюцію продукту, починаючи з простого прототипу та поступово додаючи нові функції та вдосконалюючи існуючі.

Основні етапи еволюційної моделі включають:

1. **Здійснення базового функціоналу:**
   * Створення базового прототипу або початкової версії продукту, яка включає основні функції та можливості.
2. **Оцінка та відгуки користувачів:**
   * Представлення прототипу користувачам для оцінки та отримання відгуків.
3. **Аналіз та уточнення вимог:**
   * Аналіз відгуків та коригування вимог до програми для врахування побажань користувачів.
4. **Розширення та вдосконалення:**
   * Додавання нових функцій та можливостей, вдосконалення дизайну та функціоналу.
5. **Повторення ітерацій:**
   * Повторення циклу створення, оцінки, аналізу та вдосконалення прототипу доти, поки не буде досягнуто повного задоволення від продукту.

Основні особливості еволюційної моделі включають:

* **Поступовий розвиток:** Продукт розвивається крок за кроком, додаванням нових функцій та виправленням недоліків.
* **Адаптабельність:** Можливість змінювати напрямок розробки відповідно до вимог та змін у вимаганнях користувачів.
* **Багаторазове вдосконалення:** Продукт може бути вдосконалюваний ітеративно, доки не досягне свого остаточного стану.
* **Залучення користувачів:** Активне залучення користувачів у процес визначення та вдосконалення вимог.
* **Гнучкість у визначенні функціональності:** Можливість додавання та зміни функціональності в процесі розробки.

Еволюційна модель є ефективною для проектів, де важко визначити повні вимоги наперед і коли є необхідність швидко виходити на ринок з базовим продуктом, щоб отримати відгуки від користувачів та партнерів.

7. Моделі життєвого циклу програми. Ітеративна й інкрементна модель

Ітеративна та інкрементна моделі є двома відмінними, але часто використовуваними підходами в життєвому циклі розробки програмного забезпечення.

**Ітеративна модель:**

Ітеративна модель передбачає повторення етапів розробки, дозволяючи поступово уточнювати та вдосконалювати систему. Кожна ітерація представляє собою повний цикл життєвого циклу, включаючи аналіз, проектування, реалізацію та тестування. Кожна нова ітерація додає нові функції або вдосконалює існуючі, забезпечуючи більш повністю функціональний продукт на кожному етапі.

Основні особливості ітеративної моделі:

* **Гнучкість:** Здатність вносити зміни та вдосконалення на кожній ітерації.
* **Залучення користувачів:** Взаємодія з користувачами на ранніх етапах розробки.
* **Більш ранні результати:** Можливість отримувати частковий продукт швидше.

**Інкрементна модель:**

Інкрементна модель передбачає поетапний випуск функціональних блоків або "інкрементів" програми. Кожен інкремент є повноцінним продуктом, але без повного набору функціональності. Кожен новий інкремент додає нові функції або розширює функціональність попереднього інкременту.

Основні особливості інкрементної моделі:

* **Поступовий випуск функціональності:** Система розробляється шляхом послідовного випуску функціональних блоків.
* **Тестування і відгуки:** Можливість тестувати та отримувати відгуки поетапно.
* **Швидке впровадження:** Можливість швидкого введення базового функціоналу.

Обидві ці моделі підтримують гнучкий та поетапний підхід до розробки, що дозволяє адаптуватися до змін вимог та швидко виходити на ринок із базовим продуктом. Ітеративність та інкрементність часто використовуються разом для досягнення більшої гнучкості та швидкості у розробці програмного забезпечення.

Початок форми

8. Моделі життєвого циклу програми. Модель на основі раніше створених компонент.

Модель на основі раніше створених компонент (Component-Based Development, CBD) - це підхід до розробки програмного забезпечення, де використовуються готові компоненти або модулі для побудови системи. Ця модель базується на ідеї використання вже існуючих та перевірених компонентів для створення нового програмного продукту. Вона сприяє перевикористанню коду, зменшує час розробки та підвищує надійність системи.

Основні етапи моделі на основі раніше створених компонент:

1. **Вибір компонентів:**
   * Визначення необхідних функціональних і нефункціональних вимог та вибір компонентів, які можна використовувати.
2. **Реферування компонентів:**
   * Адаптація та інтеграція обраних компонентів для відповідності конкретним потребам проекту.
3. **Створення нового функціоналу:**
   * Додавання нових функцій та компонентів для задоволення вимог конкретного проекту.
4. **Тестування:**
   * Проведення тестування для перевірки якості та взаємодії між компонентами.
5. **Поступовий розвиток:**
   * Повторення цього циклу для додавання нового функціоналу або виправлення помилок.

Основні переваги моделі на основі раніше створених компонент:

* **Швидкість розробки:** Застосування готових компонентів дозволяє значно зменшити час розробки.
* **Перевикористання:** Можливість використовувати компоненти в різних проектах.
* **Надійність:** Компоненти, які вже пройшли тестування та використовуються в інших системах, мають високий рівень надійності.
* **Підтримка змін:** Здатність легко вносити зміни та доповнення, додаючи або замінюючи компоненти.

Однак існують і виклики при використанні цієї моделі, такі як необхідність докладного вивчення та розуміння вибраних компонентів, а також управління залежностями між компонентами.

9. Моделі життєвого циклу програми. Модель формальної розробки.

Модель формальної розробки є підходом до розробки програмного забезпечення, де великий акцент приділяється математичній формалізації специфікацій, дизайну та верифікації програми. Основна ідея полягає у тому, щоб використовувати математичні методи та формальні мови для опису системи, забезпечуючи точні та безсуперечні специфікації та конструкції програми.

Основні етапи моделі формальної розробки:

1. **Специфікація:**
   * Формалізація вимог та функціональності системи за допомогою математичних та логічних засобів. Це може включати визначення пропозицій логіки, алгебр, теорії множин тощо.
2. **Аналіз та верифікація:**
   * Математичний аналіз та перевірка відповідності програми вимогам, визначеним на етапі специфікації. Це може включати формальну верифікацію, докази коректності та інші методи.
3. **Дизайн:**
   * Розробка математичного або формального опису архітектури програми на основі визначених специфікацій.
4. **Реалізація:**
   * Перетворення формального дизайну в код програми, який відповідає визначеним специфікаціям.
5. **Тестування:**
   * Використання формальних методів для тестування та перевірки правильності програми.
6. **Підтримка та розвиток:**
   * Забезпечення формальної коректності та розширення програми за необхідності.

Основні особливості моделі формальної розробки:

* **Висока точність:** Формальні методи гарантують високий рівень точності та відповідності визначеним специфікаціям.
* **Формальна верифікація:** Можливість проводити формальні докази коректності програми.
* **Складний процес:** Вимагає високого рівня експертизи та математичних знань в розробці та аналізі програм.
* **Довгий термін розробки:** Може вимагати більше часу та ресурсів у порівнянні з іншими моделями.

Ця модель використовується в основному для розробки критично важливих систем, де важлива формальна коректність та відповідність вимогам.

Початок форми

10. Постійні та змінні вимоги до програмного забезпечення.

Вимоги до програмного забезпечення можна класифікувати на постійні (стабільні) та змінні (динамічні). Ця класифікація допомагає визначити ступінь змін вимог протягом життєвого циклу розробки програмного продукту.

### Постійні вимоги (стабільні):

1. **Основні функціональні вимоги:**
   * Вимоги, які є основою для функціональності програми та залишаються стабільними протягом більшості проекту.
2. **Нефункціональні вимоги:**
   * Вимоги до якості, продуктивності, безпеки та інші, які мають сталі характеристики і визначають стандарти, яким повинен відповідати продукт.
3. **Архітектурні вимоги:**
   * Опис структури програмного продукту та зв'язків між його компонентами.
4. **Сумісність:**
   * Вимоги до взаємодії програми з іншими системами чи обладнанням.

Постійні вимоги є фундаментом розробки та вони не піддаються значним змінам під час проекту.

### Змінні вимоги (динамічні):

1. **Функціональні зміни:**
   * Додавання нових функцій чи зміна існуючих відповідно до змін потреб користувачів чи бізнес-вимог.
2. **Вимоги до продуктивності:**
   * Зміни у вимогах до продуктивності відповідно до збільшення обсягу даних чи користувацької активності.
3. **Вимоги безпеки:**
   * Зміни, пов'язані із зміною загроз та вимог щодо безпеки інформації.
4. **Вимоги до інтерфейсів:**
   * Зміни у вимогах до інтеграції з іншими системами чи інтерфейсів з користувачем.

Змінні вимоги можуть виникнути через зміну вимог користувачів, зміни в бізнес-середовищі або через еволюцію самого продукту.

Управління і збереженням балансу між постійними та змінними вимогами є важливим елементом ефективного управління проектом програмного забезпечення. Використання гнучких методологій розробки, таких як Agile, може полегшити обробку змін у вимогах під час розробки продукту.

Початок форми

11. Функціональні вимоги до програмного забезпечення.

Функціональні вимоги (Functional Requirements) є частиною специфікацій для програмного забезпечення, які визначають, які функції та операції має виконувати система. Ці вимоги описують, як система повинна взаємодіяти з користувачем, іншими системами та середовищем. Основна мета функціональних вимог - забезпечити конкретну функціональність та можливості програмного продукту.

Основні компоненти функціональних вимог включають:

1. **Опис функцій:**
   * Конкретне визначення функціональності системи, що включає в себе можливості, які система повинна надавати користувачам чи іншим системам.
2. **Введення та виведення даних:**
   * Опис того, які дані система отримує від користувачів чи інших систем, та які дані вона генерує у вигляді виведення.
3. **Поведінка системи:**
   * Опис реакцій системи на конкретні входи та умови, включаючи порядок виконання операцій та послідовність подій.
4. **Обмеження та умови:**
   * Визначення умов, за яких система повинна функціонувати, а також обмежень та обов'язкові умови використання.
5. **Сценарії використання:**
   * Опис конкретних ситуацій чи сценаріїв, в яких користувач взаємодіє з системою та очікує певних результатів.
6. **Вимоги до продуктивності:**
   * Вимоги, пов'язані з продуктивністю системи, такі як швидкодія, завантаження та обробка даних.
7. **Вимоги до безпеки:**
   * Визначення функцій та механізмів для забезпечення безпеки програмного продукту.

Функціональні вимоги визначають "що" система повинна робити, а не "як". Для їх формалізації часто використовуються спеціальні методи та мови, такі як Use Case, діаграми активностей, або формальні мови програмування.

Приклад функціональної вимоги: "Система повинна дозволяти користувачеві авторизуватися за допомогою електронної пошти та пароля."

12. Нефункціональні вимоги до програмного забезпечення.

Нефункціональні вимоги (Non-Functional Requirements) визначають атрибути якості та обмеження, які не стосуються конкретної функціональності системи, але визначають якість, ефективність, безпеку та інші характеристики програмного забезпечення. Ці вимоги визначають умови експлуатації системи та її середовище.

Основні категорії нефункціональних вимог включають:

1. **Продуктивність:**
   * *Час відгуку:* Максимальний час, за який система повинна реагувати на запит користувача.
   * *Продуктивність системи:* Максимальне число операцій чи транзакцій, які система повинна обробляти протягом певного періоду часу.
   * *Завантаження системи:* Максимальне число користувачів, які можуть використовувати систему одночасно.
2. **Надійність:**
   * *Наявність:* Процент часу, протягом якого система повинна бути доступною для користувачів.
   * *Стійкість:* Здатність системи витримувати відмови та відновлювати свою роботу.
3. **Безпека:**
   * *Керування доступом:* Система повинна забезпечувати захищений доступ до функцій та даних від несанкціонованих користувачів.
   * *Шифрування даних:* Вимоги до шифрування даних для забезпечення конфіденційності.
4. **Споживана пам'ять та ресурси:**
   * *Обсяг використаної пам'яті:* Максимальний обсяг оперативної пам'яті, який система може використовувати.
   * *Використання процесора:* Максимальне використання обчислювальних ресурсів системою.
5. **Сумісність:**
   * *Сумісність з іншими системами:* Можливість взаємодії системи з іншими програмами чи обладнанням.
6. **Інтерфейс та користувацький досвід:**
   * *Ергономіка та дизайн:* Вимоги до інтерфейсу, що забезпечують зручність та естетичний дизайн.
   * *Мовна підтримка:* Підтримка певної мови чи набору мов.
7. **Управління та підтримка:**
   * *Можливість розширення:* Здатність системи до зміни та розширення функціональності.
   * *Документація:* Вимоги до документації, що включає інструкції користувача, технічну документацію та інше.

Ці нефункціональні вимоги допомагають визначити загальні характеристики та умови експлуатації системи та є важливим елементом при визначенні повного спектру вимог до програмного продукту.

13. Аналіз, систематизація, специфікація та формалізація вимог до програмного забезпечення.

Аналіз, систематизація, специфікація та формалізація вимог до програмного забезпечення - це важливі етапи у розробці програмних продуктів, які допомагають зрозуміти та чітко визначити очікування від системи. Нижче подано короткий огляд кожного з цих етапів:

### 1. Аналіз вимог:

Аналіз вимог включає в себе вивчення та розуміння потреб користувачів та бізнес-вимог до системи. Цей етап включає в себе збір інформації, виявлення основних функціональних та нефункціональних вимог, а також визначення обмежень та умов роботи системи.

### 2. Систематизація вимог:

Систематизація вимог передбачає організацію та групування вимог за різними категоріями чи темами. Це полегшує подальший аналіз та визначення пріоритетів, а також робить вимоги більш структурованими.

### 3. Специфікація вимог:

Специфікація вимог - це процес детального визначення кожної вимоги, щоб забезпечити її однозначність та розуміння. Специфікація може включати в себе текстові описи, діаграми, таблиці та інші засоби, щоб чітко визначити, як система повинна поводитися у визначених ситуаціях.

### 4. Формалізація вимог:

Формалізація вимог передбачає використання формальних методів та мов для математичного визначення вимог. Це може включати в себе використання формальних мов програмування, логічних висловлювань, алгебр та інших математичних засобів для точного визначення функціональності та характеристик системи.

Користь формалізації полягає в тому, що вона зменшує ризик непорозумінь та допомагає уникнути двозначностей у сприйнятті вимог. Однак цей підхід може бути складним і вимагає високого рівня експертизи в області формальних методів.

Загальний процес аналізу, систематизації, специфікації та формалізації вимог є ітеративним, і вимагає взаємодії між розробниками, замовниками та іншими зацікавленими сторонами для забезпечення повного та зрозумілого опису системи.

14. Верифікація та атестація програмного забезпечення.

Верифікація та атестація є важливими етапами в процесі розробки та впровадження програмного забезпечення. Ці терміни вказують на різні аспекти перевірки та підтвердження якості та відповідності програмного продукту вимогам та стандартам.

### Верифікація:

**Верифікація** - це процес перевірки того, чи відповідає програмне забезпечення визначеним специфікаціям і вимогам. Основна мета верифікації - переконатися, що розроблене програмне забезпечення функціонує вірно та відповідає вимогам, встановленим на етапі аналізу та проектування. Верифікація може включати такі дії, як кодування, тестування, аналіз та перевірку документації.

**Етапи верифікації:**

1. **Аналіз вимог:** Перевірка правильності та повноти визначених вимог.
2. **Дизайн:** Перевірка архітектури та дизайну програми на відповідність вимогам.
3. **Кодування:** Перевірка написаного коду на відповідність специфікаціям та стандартам.
4. **Тестування:** Перевірка, що програмне забезпечення працює правильно та виконує функції відповідно до очікувань.
5. **Аналіз документації:** Перевірка наявності та коректності документів, які описують програму.

### Атестація:

**Атестація** - це процес підтвердження, що програмне забезпечення відповідає певним стандартам, вимогам та призначенню. Атестація визначається формальним документом (атестатом), який свідчить про те, що програма відповідає визначеним критеріям. Атестацію може проводити внутрішній або зовнішній орган, а в деяких галузях вона може бути обов'язковою (наприклад, у випадку розробки програмного забезпечення для медичних пристроїв чи банківських систем).

**Етапи атестації:**

1. **Підготовка:** Збір та підготовка всіх необхідних документів та інформації для атестації.
2. **Внутрішнє тестування:** Перевірка відповідності програмного продукту внутрішнім стандартам та вимогам.
3. **Зовнішнє тестування:** Перевірка на відповідність зовнішнім стандартам та вимогам.
4. **Аудит:** Перевірка відповідності процесів розробки та тестування стандартам та нормам.
5. **Оформлення атестату:** Видання офіційного атестату, який підтверджує, що програмне забезпечення відповідає вимогам.

Важливо враховувати, що верифікація та атестація - це взаємодоповнюючі процеси, спрямовані на забезпечення якості та відповідності програмного забезпечення визначеним стандартам та вимогам.

Початок форми

15. Концепції підтримки користувачів за допомогою інтерфейсів.

Підтримка користувачів через інтерфейси — це ключовий аспект забезпечення задоволення та ефективного використання продуктів або послуг. Інтерфейси для підтримки користувачів можуть бути реалізовані в різних формах. Ось кілька концепцій, які використовуються для забезпечення ефективної підтримки:

1. **Віртуальні асистенти та чат-боти:**
   * *Опис:* Використання штучного інтелекту для створення програм, які можуть взаємодіяти з користувачами через текстові чи голосові інтерфейси.
   * *Переваги:* Розподілення інформації, вирішення типових запитань, автоматизована підтримка.
   * *Приклади:* Siri, Google Assistant, чат-боти на веб-сайтах.
2. **Системи тікетів та електронна пошта:**
   * *Опис:* Засновані на веб-сервісах та електронній пошті, ці інтерфейси дозволяють користувачам створювати та відстежувати тікети для розв'язання проблем або отримання допомоги.
   * *Переваги:* Систематизація та відстеження запитань, можливість подальшої обробки та аналізу.
   * *Приклади:* Zendesk, Freshdesk, системи обробки тікетів у великих компаніях.
3. **Відеозв'язок та онлайн-конференції:**
   * *Опис:* Використання відеозв'язку для безпосереднього з'єднання користувачів з представниками служби підтримки через інтернет.
   * *Переваги:* Персональне спілкування, можливість демонстрації та розв'язання проблем на відстані.
   * *Приклади:* Zoom, Microsoft Teams, відеозв'язок на веб-сайтах підтримки.
4. **Самообслуговування та бази знань:**
   * *Опис:* Надання користувачам доступу до документації, FAQ, інструкцій та баз знань для самостійного пошуку відповідей на запитання.
   * *Переваги:* Швидкий доступ до інформації, ефективне вирішення стандартних питань.
   * *Приклади:* Внутрішні бази знань компаній, онлайн-документація.
5. **Соціальні мережі та форуми:**
   * *Опис:* Спільноти на платформах соціальних мереж або форуми, де користувачі можуть ділитися досвідом та отримувати підтримку від інших користувачів або представників компанії.
   * *Переваги:* Взаємодія користувачів, можливість швидкої відповіді.
   * *Приклади:* Reddit, Stack Overflow, форуми підтримки.

Ці концепції можуть використовуватися окремо або комбінуватися для створення комплексної системи підтримки користувачів, яка забезпечить ефективну та доступну допомогу.

16. Види документації до програмного забезпечення.

Документація до програмного забезпечення відіграє важливу роль у забезпеченні зрозумілості, підтримки та розвитку програмних продуктів. Різні види документації адресують різні аспекти розробки, використання та підтримки програмного забезпечення. Ось деякі типи документації:

1. **Технічна документація:**
   * **Специфікація вимог (Requirements Specification):** Опис вимог до програмного забезпечення.
   * **Дизайн-документація (Design Documentation):** Опис архітектури, модулів та інших деталей проекту.
   * **Технічний опис (Technical Documentation):** Детальний опис коду, алгоритмів та структур даних.
   * **API-документація (API Documentation):** Інструкції для розробників щодо використання програмного інтерфейсу.
2. **Експлуатаційна документація:**
   * **Інструкції користувача (User Manuals):** Керівництва з користування програмою.
   * **Посібники з встановлення та налаштування (Installation and Configuration Guides):** Інструкції з встановлення та налаштування продукту.
   * **Документація з оновлень (Release Notes):** Інформація про зміни у нових версіях програми.
   * **Техпідтримка (Troubleshooting Guides):** Інструкції для вирішення проблем та усунення неполадок.
3. **Документація з проектування та стратегії:**
   * **Плани проекту (Project Plans):** Документи, які визначають мети, завдання та розподіл ресурсів.
   * **Документи стратегії розробки (Development Strategy Documents):** Детальні плани розробки та випуску продукту.
   * **Документація з тестування (Testing Documentation):** Плани тестування, тестові сценарії та результати тестування.
4. **Документація з безпеки:**
   * **Політика безпеки (Security Policy):** Визначення правил та процедур безпеки.
   * **Аналіз загроз (Threat Analysis):** Опис потенційних загроз безпеці програмного продукту та способів їх запобігання.
   * **Інструкції з безпеки (Security Guidelines):** Рекомендації для користувачів щодо забезпечення безпеки при використанні програмного продукту.
5. **Документація з ліцензування та правова документація:**
   * **Ліцензійна угода (End User License Agreement - EULA):** Угода, що визначає умови використання програми.
   * **Правила конфіденційності (Privacy Policy):** Політика щодо збору та використання персональної інформації користувачів.
   * **Юридичні угоди (Legal Agreements):** Інші юридичні документи, які можуть бути важливими для користувачів та розробників.

Ці види документації допомагають розробникам, користувачам та іншим зацікавленим сторонам зрозуміти, використовувати, підтримувати та розвивати програмне забезпечення.

17. Складання та розрахунок плану проекту.

Складання та розрахунок плану проекту — це ключовий етап у процесі управління проектом. Ефективний план проекту допомагає здійснити планування, розподіл ресурсів, визначення термінів виконання завдань та встановлення пріоритетів. Нижче подано загальні кроки та важливі аспекти при складанні та розрахунку плану проекту:

### Крок 1: Визначення мети та об'єму проекту

1. **Визначте мету проекту:** Чітко сформулюйте ціль та об'єктиви проекту.
2. **Розкрийте об'єм проекту:** Визначте, що входить та не входить в об'єм проекту.

### Крок 2: Визначення завдань та робочих пакетів

1. **Розбийте проект на завдання:** Розгляньте всі етапи та завдання, які потрібно виконати.
2. **Створіть робочі пакети:** Розподіліть завдання на менші, керовані блоки (робочі пакети).

### Крок 3: Оцінка та призначення ресурсів

1. **Оцініть тривалість завдань:** Визначте, скільки часу потрібно для виконання кожного завдання.
2. **Призначте ресурси:** Визначте, хто із команди або зовнішніх джерел буде відповідальний за виконання кожного завдання.

### Крок 4: Розклад завдань

1. **Побудуйте сітку завдань:** Використовуйте методи, такі як Ганта чи Pert, для визначення послідовності та залежностей завдань.
2. **Складіть графік:** Враховуючи тривалість та залежності, створіть графік виконання завдань.

### Крок 5: Розрахунок критичного шляху

1. **Визначте критичний шлях:** Знайдіть послідовність завдань, які визначають тривалість проекту.
2. **Визначте терміни та здійсніть розрахунки:** З'ясуйте, коли кожне завдання повинно бути виконане та розрахуйте часові резерви.

### Крок 6: Розрахунок вартості

1. **Оцініть бюджет:** Розгляньте вартість ресурсів, обладнання та інші витрати, пов'язані з проектом.
2. **Створіть бюджет проекту:** Призначте кошти на різні завдання та фази проекту.

### Крок 7: Розробка системи контролю та звітності

1. **Визначте систему контролю:** Створіть механізми відстеження виконання завдань та витрат.
2. **Підготуйте звіти:** Визначте, як часто та які дані будуть включені в звіти для стейкхолдерів.

### Крок 8: Розробка плану ризиків

1. **Визначте потенційні ризики:** Ідентифікуйте можливі проблеми, які можуть виникнути під час виконання проекту.
2. **Створіть план управління ризиками:** Розробіть стратегії для запобігання, мінімізації та вирішення ризиків.

### Крок 9: Затвердження та відстеження плану

1. **Затвердіть план:** Отримайте погодження та затвердження плану від усіх зацікавлених сторін.
2. **Відстежуйте виконання:** Постійно оновлюйте та відстежуйте виконання проекту за допомогою системи контролю.

### Загальні рекомендації:

* **Комунікація:** Забезпечте ефективну комунікацію між усіма учасниками проекту.
* **Гнучкість:** Періодично переглядайте та адаптуйте план, особливо при змінах обставин.

Важливо пам'ятати, що план проекту — це живий документ, який може змінюватися. Його необхідно постійно вдосконалювати та адаптувати до реальних обставин і змін у проекті.

18. Керування ризиками при розробці проекту.

Керування ризиками є важливою складовою процесу управління проектом. Ризики можуть виникнути через невизначеність, непередбачувані обставини або помилки у процесі розробки. Ефективне управління ризиками дозволяє підготуватися до можливих негативних подій та зменшити їх вплив на проект. Ось кілька кроків для керування ризиками при розробці проекту:

### 1. ****Ідентифікація ризиків:****

* **Створення команди для ідентифікації ризиків:** Залучіть членів команди та зацікавлених сторін для виявлення потенційних ризиків.
* **Аналіз областей ризиків:** Розділіть проект на фази та області, щоб визначити потенційні ризики в кожній з них.
* **Використання досвіду:** Погляньте на аналогічні проекти або ситуації, щоб виявити та вивчити можливі ризики.

### 2. ****Аналіз ризиків:****

* **Оцінка ймовірності та впливу:** Визначте ймовірність виникнення кожного ризику та його можливий вплив на проект.
* **Побудова матриці ризиків:** Створіть матрицю ризиків для ідентифікації та класифікації ризиків за їхнім впливом та ймовірністю.

### 3. ****Розробка стратегій управління ризиками:****

* **Уникнення:** Уникайте ситуацій, які можуть викликати ризик.
* **Прийняття:** Приймайте ризик і готуйтеся до нього.
* **Передача:** Передавайте частину або всі ризики стороннім агентам, наприклад, за допомогою страхування.
* **Зменшення:** Приймайте заходи для зменшення ймовірності або впливу ризику.

### 4. ****Впровадження стратегій:****

* **Розробка плану реагування на ризики:** Для кожного ідентифікованого ризику створіть план дій у випадку його виникнення.
* **Проведення тренувань та симуляцій:** Перевірте ефективність стратегій та планів реагування на ризики під час тренувань.

### 5. ****Відстеження та контроль ризиків:****

* **Постійний моніторинг:** Слідкуйте за змінами у середовищі проекту та вчасно виявляйте нові ризики.
* **Оновлення плану управління ризиками:** Регулярно оновлюйте стратегії та плани реагування на ризики відповідно до змін у проекті.
* **Вивчення здійснених дій:** Аналізуйте, як ризики виникали та як їх вирішували, для покращення стратегій управління.

### 6. ****Звітність та комунікації:****

* **Звіти стейкхолдерам:** Забезпечте інформацією стейкхолдерів щодо ідентифікованих, відстежуваних та управлінних ризиків.
* **Регулярна комунікація:** Підтримуйте відкриту комунікацію щодо ризиків як серед команди, так і з іншими зацікавленими сторонами.

Управління ризиками — це постійний та ітеративний процес. Зосередьтеся на запобіганні, адаптації та вчасному реагуванні для забезпечення успішного виконання проекту.

19. Методи оцінювання програмного забезпечення.

Оцінювання програмного забезпечення включає в себе різні методи та інструменти для визначення якості, ефективності та інших аспектів програмного продукту. Ось деякі з методів оцінювання програмного забезпечення:

1. **Тестування програмного забезпечення:**
   * **Функціональне тестування:** Перевірка, чи виконує програмне забезпечення визначені функції згідно специфікацій.
   * **Навантажувальне тестування:** Оцінка роботи системи під навантаженням та визначення її максимальних можливостей.
   * **Тестування безпеки:** Виявлення та виправлення потенційних вразливостей та загроз безпеці програмного продукту.
   * **Автоматизоване тестування:** Використання програм для виконання тестів та автоматизації процесу тестування.
2. **Оцінка продуктивності:**
   * **Вимірювання часу відгуку:** Визначення часу, необхідного для виконання конкретної операції.
   * **Моніторинг використання ресурсів:** Спостереження за використанням ресурсів (пам'яті, процесора, дискового простору) під час роботи програмного забезпечення.
   * **Вимірювання пропускної здатності:** Визначення кількості операцій, які система може обробити за певний період часу.
3. **Оцінка користувальницького досвіду:**
   * **Оцінка зручності використання (Usability Testing):** Визначення того, наскільки легко користувачі можуть взаємодіяти з програмним продуктом.
   * **Збір фідбеку від користувачів:** Отримання відгуків та реакцій користувачів на різні аспекти програмного забезпечення.
4. **Аудит програмного забезпечення:**
   * **Технічний аудит коду:** Аналіз якості та безпеки програмного коду.
   * **Аудит безпеки:** Оцінка відповідності програмного продукту стандартам безпеки та виявлення можливих ризиків.
5. **Вимірювання якості коду:**
   * **Метрики програмного коду:** Використання різних метрик для визначення якості коду, таких як кількість рядків коду, рівень вкладеності, кількість помилок тощо.
   * **Статичний аналіз коду:** Аналіз програмного коду без його виконання для виявлення потенційних проблем.
6. **Оцінка вартості та продуктивності:**
   * **Вартісна оцінка:** Розрахунок фінансових витрат на розробку та підтримку програмного продукту.
   * **Визначення ROI (Return on Investment):** Оцінка прибутковості інвестицій у розробку програмного забезпечення.

Ці методи можна використовувати як окремо, так і в комбінації для отримання повного обсягу оцінки якості та ефективності програмного забезпечення.

20. Керування якістю програмного забезпечення.

Керування якістю програмного забезпечення (Quality Management in Software Engineering) — це процеси та практики, спрямовані на забезпечення того, що програмний продукт відповідає визначеним стандартам якості та вимогам користувачів. Ефективне керування якістю включає в себе різні етапи та методи, щоб переконатися, що програмне забезпечення відповідає встановленим критеріям якості. Ось деякі ключові аспекти керування якістю програмного забезпечення:

### 1. ****Визначення Вимог до Якості:****

* Визначення та документування конкретних вимог до якості, які повинні бути враховані під час розробки.
* Встановлення критеріїв прийняття, які дозволяють визначити, коли програмне забезпечення вважається задовільним.

### 2. ****Управління Версіями та Конфігурацією:****

* Забезпечення ефективного контролю версій та конфігурацій програмного забезпечення.
* Здійснення систематичного відслідковування змін та документація конфігураційних елементів.

### 3. ****Стандарти та Рекомендації:****

* Визначення та використання стандартів розробки для забезпечення відповідності до визначених норм та правил.
* Впровадження рекомендацій та кращих практик у розробку.

### 4. ****Тестування та Валідація:****

* Здійснення систематичних тестів для визначення відповідності продукту вимогам до якості.
* Валідація функціональності та визначення коректності роботи програмного забезпечення.

### 5. ****Аудит та Огляди Коду:****

* Використання аудиту та оглядів коду для виявлення потенційних помилок та невідповідностей стандартам.
* Систематичне переглядання коду командою розробників для забезпечення якості виконання завдань.

### 6. ****Управління Ризиками:****

* Визначення потенційних ризиків, які можуть вплинути на якість програмного продукту.
* Розробка стратегій управління ризиками та систематичне відслідковування ризиків під час проекту.

### 7. ****Неперервна Інтеграція та Інтеграційні Тести:****

* Використання процесу неперервної інтеграції для регулярної перевірки взаємодії компонентів програмного забезпечення.
* Здійснення інтеграційних тестів для визначення коректності спільної роботи компонентів.

### 8. ****Консистентність та Забезпечення Якості Коду:****

* Вимагання дотримання стандартів коду та використання консистентного стилю програмування.
* Використання інструментів для автоматичної перевірки якості коду.

### 9. ****Технічна Підтримка та Відгуки Користувачів:****

* Забезпечення належної технічної підтримки для виявлення та виправлення помилок.
* Збір відгуків від користувачів та використання їх для подальшого вдосконалення продукту.

### 10. ****Оцінка Продуктивності:****

* Спостереження та оцінка продуктивності програмного забезпечення в реальних умовах експлуатації.
* Здійснення аналізу продуктивності та виявлення можливих точок оптимізації.

### 11. ****Навчання та Підвищення Кваліфікації:****

* Забезпечення навчання розробників та членів команди в сучасних методах та інструментах розробки.
* Сприяння професійному росту та обмін досвідом в команді.

Загальна мета керування якістю програмного забезпечення полягає в тому, щоб забезпечити, що програмний продукт відповідає вимогам замовника, є стійким та забезпечує задоволення користувачів.

Початок форми

21. Метрики програмного забезпечення.

Метрики програмного забезпечення — це кількісні показники, які використовуються для вимірювання різних характеристик процесу розробки, якості та ефективності програмного продукту. Метрики допомагають визначити стан проекту, виявити можливі проблеми та вдосконалити процеси розробки. Ось деякі ключові метрики програмного забезпечення:

### 1. ****Метрики Процесу Розробки:****

* **Продуктивність Розробників (Productivity):** Кількість робочих одиниць, що створює розробник за одиницю часу.
* **Швидкість Доставки (Delivery Speed):** Середня тривалість часу від початку розробки до видачі готового продукту.
* **Час Виявлення Помилок (Time to Detect Defects):** Середній час, необхідний для виявлення та виправлення помилок.
* **Частота Випуску (Release Frequency):** Кількість випусків програмного продукту за певний період.

### 2. ****Метрики Коду та Дизайну:****

* **Покриття Тестами (Test Coverage):** Процент коду, який покритий тестами.
* **Складність Коду (Code Complexity):** Міра складності програмного коду, часто вимірюється за допомогою метрик, таких як Cyclomatic Complexity.
* **Співвідношення Кількості Помилок до Рядка Коду (Defect Density):** Кількість помилок на одиницю коду.
* **Кількість Повторень Коду (Code Duplication):** Визначення кількості дубльованого коду в системі.

### 3. ****Метрики Якості Програмного Продукту:****

* **Стабільність (Reliability):** Ймовірність того, що програмний продукт виконає свої функції без збоїв.
* **Ефективність (Efficiency):** Використання ресурсів системи, таких як пам'ять та процесор.
* **Надійність (Maintainability):** Легкість, з якою можна внести зміни у програмний код та додавати новий функціонал.
* **Продуктивність (Performance):** Швидкість та ефективність роботи програмного продукту при різних навантаженнях.

### 4. ****Метрики Вартості та Бізнес-Вигод:****

* **Вартість Розробки (Development Cost):** Загальна вартість розробки програмного продукту.
* **Прибутковість (Profitability):** Визначення того, наскільки успішно програмний продукт генерує прибуток для бізнесу.
* **Вартість Підтримки (Support Cost):** Загальна вартість підтримки та обслуговування програмного продукту.
* **Повернення Інвестицій (Return on Investment - ROI):** Відношення прибутку до вартості інвестицій.

### 5. ****Метрики Користувачів та Задоволення Клієнтів:****

* **Задоволеність Користувачів (User Satisfaction):** Оцінка та відгуки користувачів щодо задоволеності продуктом.
* **Кількість Запитань Техпідтримки (Support Tickets):** Кількість запитань та проблем, які користувачі висловлюють у службу підтримки.
* **Активність Користувачів (User Engagement):** Метрики, такі як частота використання, тривалість сесій та інші, які вказують на активність користувачів.

Ці метрики допомагають командам розробників та менеджерам визначати, які аспекти процесу розробки та якості програмного продукту потребують уваги та вдосконалення. Важливо враховувати, що вибір метрик повинен бути адаптований до конкретних потреб та характеристик проекту.

22. Основні компоненти та призначення діаграми варіантів використання. Навести приклад.

**Діаграма варіантів використання (Use Case Diagram):** Це графічний інструмент моделювання, який використовується для візуалізації, опису та спілкування стосовно функціональності системи або програмного продукту. Основними компонентами діаграми варіантів використання є актори та варіанти використання.

### Основні компоненти:

1. **Актори (Actors):**
   * **Актори представляють ролі, які взаємодіють із системою.**
   * **Це можуть бути користувачі, зовнішні системи чи інші системні елементи.**
   * **Актори не є часткою системи, вони взаємодіють з нею.**
2. **Варіанти використання (Use Cases):**
   * **Варіанти використання представляють конкретні функціональні можливості або дії системи.**
   * **Кожен варіант використання вказує на те, як система взаємодіє з актором для досягнення конкретної мети.**
3. **Система (System):**
   * **Зона, яка оточує всі актори та варіанти використання.**
   * **Представляє собою межі системи, яку ви моделюєте.**

### Призначення:

1. **Візуалізація Функціональності:**
   * **Діаграма варіантів використання дозволяє візуалізувати, як користувачі або зовнішні системи будуть взаємодіяти із системою.**
2. **Опис Варіантів Використання:**
   * **Для кожного варіанту використання можна надати детальний опис, який вказує на послідовність дій та умови.**
3. **Спілкування та Зрозуміння:**
   * **Діаграма є ефективним інструментом для комунікації між різними учасниками проекту, такими як розробники, менеджери та замовники.**
4. **Визначення Границь Системи:**
   * **Допомагає визначити, які частини функціональності системи входять до її області відповідальності.**

### Приклад:

Нехай ми маємо систему для онлайн-магазину. Декілька компонентів діаграми варіантів використання може виглядати так:

1. **Актори:**
   * **Покупець (Customer):** Основний користувач системи, який здійснює покупки.
   * **Адміністратор (Administrator):** Особа, яка управляє та адмініструє магазин.
2. **Варіанти Використання:**
   * **Перегляд Товарів (Browse Products):** Покупець може переглядати товари в магазині.
   * **Додавання до Кошика (Add to Cart):** Покупець може додавати товари до свого кошика.
   * **Оформлення Замовлення (Checkout):** Покупець може оформляти своє замовлення та проводити оплату.
   * **Управління Товарами (Manage Products):** Адміністратор може додавати, видаляти та редагувати товари в магазині.

Ця діаграма дозволяє візуалізувати основні функціональність системи та взаємодію з акторами для різних сценаріїв використання.

23. Основні компоненти та призначення діаграми класів. Навести приклад.

**Діаграма класів (Class Diagram):** Це один із основних видів діаграм UML, який використовується для моделювання структури системи та відображення взаємозв'язків між класами. Діаграма класів містить класи, атрибути, методи та взаємозв'язки між класами. Основними компонентами діаграми класів є:

### Основні компоненти:

1. **Клас (Class):**
   * **Представляє абстракцію об'єкта або сутності, яка має певні характеристики (атрибути) та поведінку (методи).**
2. **Атрибут (Attribute):**
   * **Представляє характеристики класу або об'єкта. Як правило, зазначається ім'я та тип атрибута.**
3. **Метод (Method):**
   * **Представляє операції чи функції, які клас може виконувати.**
4. **Взаємозв'язок (Association):**
   * **Показує зв'язки між класами та вказує, як об'єкти одного класу взаємодіють з об'єктами іншого класу.**
5. **Композиція (Composition):**
   * **Вказує, що об'єкти одного класу є частинами об'єктів іншого класу і не можуть існувати окремо.**
6. **Агрегація (Aggregation):**
   * **Вказує, що об'єкти одного класу можуть існувати окремо від об'єктів іншого класу.**

### Призначення:

1. **Моделювання Структури:**
   * **Діаграма класів дозволяє моделювати структуру системи, ідентифікувати класи та їх взаємозв'язки.**
2. **Аналіз Об'єктів:**
   * **Використовується для аналізу структури об'єктів та їх характеристик (атрибутів та методів).**
3. **Опис Логічної Архітектури:**
   * **Допомагає визначити класи та їх взаємозв'язки, що входять до складу системи, що моделюється.**
4. **Дизайн Класів:**
   * **Дозволяє розробникам проектувати та визначати класи для реалізації конкретної функціональності.**
5. **Узагальнення та Спеціалізація:**
   * **Дозволяє використовувати концепції узагальнення (generalization) та спеціалізації (specialization) для моделювання відносин між класами.**



24. Основні компоненти та призначення діаграми діяльності. Навести приклад.

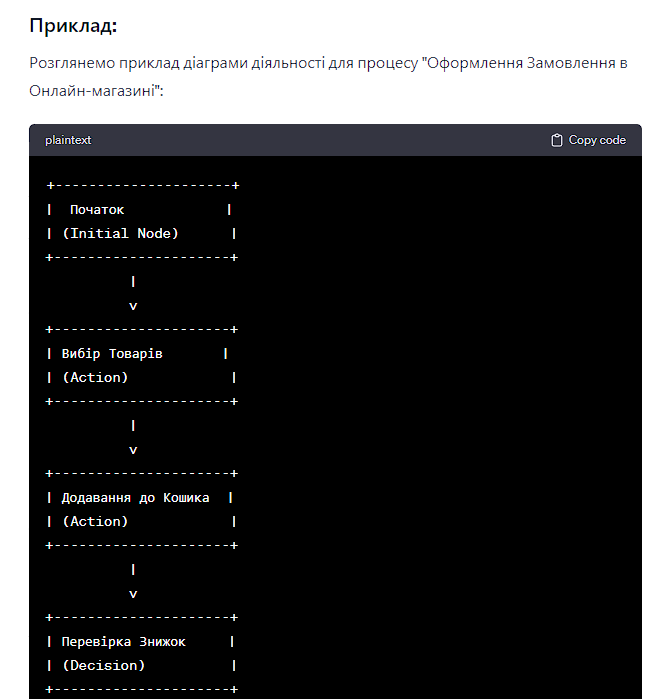
**Діаграма діяльності (Activity Diagram):** Це вид діаграми в мові моделювання UML, який використовується для візуалізації послідовності дій або процесів у системі. Діаграма діяльності дозволяє моделювати високорівневі процеси та послідовності кроків у системі. Основні компоненти діаграми діяльності включають:

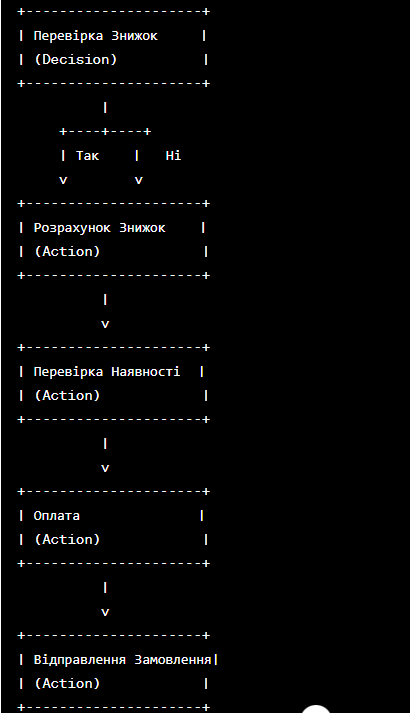
### Основні компоненти:

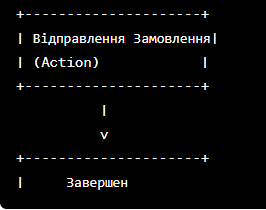
1. **Дія (Action):**
   * **Представляє конкретні дії чи активності, які виконуються в системі.**
   * **Може включати такі елементи, як виклики методів, обчислення, передача даних тощо.**
2. **Рішення (Decision):**
   * **Використовується для моделювання точок рішення у процесі, що може призвести до різних варіантів виконання.**
   * **Має умову, яка визначає, який шлях слід вибрати.**
3. **Об'єднання (Merge):**
   * **Позначає об'єднання двох або більше шляхів в процесі дії.**
4. **Виливається (Fork):**
   * **Представляє відгалуження процесу на два або більше паралельних шляхи в одному місці.**
5. **Об'єднання (Join):**
   * **Позначає об'єднання двох або більше паралельних шляхів в процесі.**
6. **Початок (Initial Node):**
   * **Позначає початок виконання процесу.**
7. **Кінець (Final Node):**
   * **Позначає завершення виконання процесу.**

### Призначення:

1. **Моделювання Послідовності Дій:**
   * **Діаграма діяльності дозволяє моделювати та візуалізувати послідовність дій або операцій у системі.**
2. **Аналіз та Визначення Процесів:**
   * **Використовується для аналізу та визначення бізнес-процесів або робочих процесів в системі.**
3. **Покращення Ефективності:**
   * **Дозволяє виявляти можливості оптимізації та покращення ефективності діяльності системи.**
4. **Робочі Процеси та Алгоритми:**
   * **Використовується для моделювання робочих процесів та алгоритмів, які можуть бути реалізовані в системі.**







25. Основні компоненти та призначення діаграми взаємодії. Навести приклад.

**Діаграма взаємодії (Interaction Diagram):** Це вид діаграми UML, який моделює взаємодію між об'єктами або ролями в рамках конкретного сценарію чи випадку використання. Діаграма взаємодії включає два основних види діаграм: діаграми послідовності та діаграми співробітництва. Основні компоненти діаграми взаємодії включають:

### Основні компоненти:

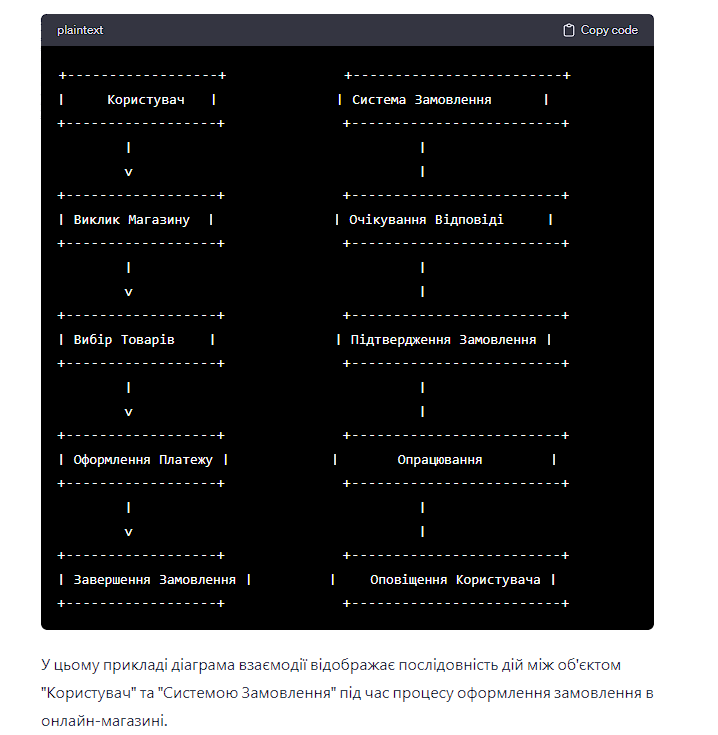
1. **Об'єкти (Objects):**
   * **Представляють конкретні екземпляри класів або елементи системи, які беруть участь у взаємодії.**
2. **Лінії Життєвого Циклу (Lifelines):**
   * **Позначають часовий інтервал існування об'єкта під час взаємодії.**
3. **Послідовність (Sequence):**
   * **Вказує послідовність повідомлень між об'єктами або ролями протягом конкретного сценарію.**
4. **Повідомлення (Message):**
   * **Вказує дії чи взаємодію між об'єктами, які відбуваються в рамках сценарію. Може бути синхронним чи асинхронним.**
5. **Виливається (Fork) та Об'єднується (Join):**
   * **Використовуються для відокремлення або об'єднання паралельних шляхів взаємодії.**

### Призначення:

1. **Моделювання Взаємодії:**
   * **Діаграма взаємодії дозволяє моделювати та візуалізувати взаємодію об'єктів або ролей в рамках конкретного сценарію чи випадку використання.**
2. **Розуміння Послідовності Дій:**
   * **Дозволяє детально розібрати послідовність дій та обмін повідомленнями між об'єктами під час виконання певного сценарію.**
3. **Аналіз Сценаріїв Використання:**
   * **Використовується для аналізу сценаріїв використання системи та визначення, як об'єкти взаємодіють у реальному часі.**
4. **Уточнення Моделей Класів:**
   * **Допомагає уточнювати моделі класів та їх взаємодії на більш високому рівні деталей.**

### Приклад:

Розглянемо приклад діаграми взаємодії для процесу "Оформлення Замовлення в Онлайн-магазині" між об'єктом "Користувач" та "Системою Замовлення":



26. Основні компоненти та призначення діаграми станів та переходів. Навести приклад.

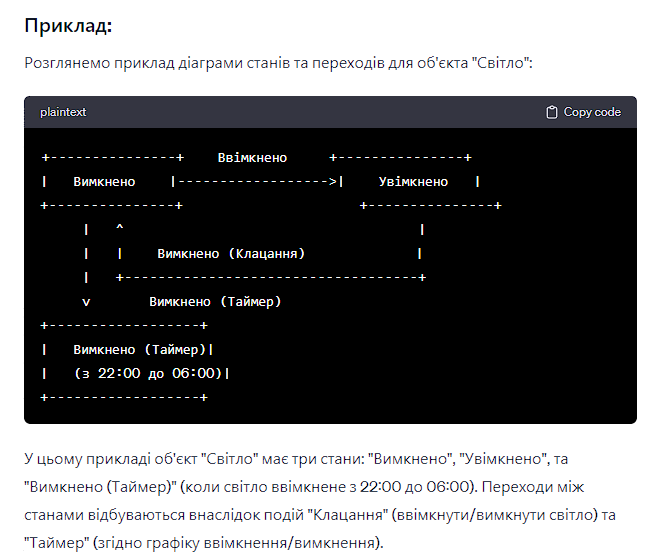
**Діаграма станів та переходів (Statechart Diagram):** Це вид діаграми UML, який використовується для моделювання станів, подій та переходів, через які система проходить під час свого життєвого циклу або відповідно до певного подійного впливу. Основні компоненти діаграми станів та переходів включають:

### Основні компоненти:

1. **Стан (State):**
   * **Позначає конкретний стан системи. Може бути абстрактним або конкретним.**
2. **Переход (Transition):**
   * **Вказує на можливий перехід системи із одного стану в інший відповідно до певних умов чи подій.**
3. **Подія (Event):**
   * **Сигнал чи спостереження, яке може спричинити перехід системи з одного стану в інший.**
4. **Дія (Action):**
   * **Описує дії, які виконуються в певних станах чи під час переходів.**
5. **Початковий стан (Initial State):**
   * **Позначає стан, в якому система перебуває під час свого старту чи ініціалізації.**
6. **Кінцевий стан (Final State):**
   * **Позначає завершення роботи системи чи її життєвого циклу.**

### Призначення:

1. **Моделювання Поведінки Об'єктів:**
   * **Діаграма станів та переходів дозволяє моделювати та візуалізувати зміни станів об'єктів чи системи відповідно до подій та умов.**
2. **Події та Впливи:**
   * **Використовується для ідентифікації подій, які можуть впливати на поведінку системи та спричиняти переходи між станами.**
3. **Життєвий Цикл Системи:**
   * **Дозволяє моделювати різні стани та переходи системи під час її життєвого циклу, відслідковувати зміни та взаємодію.**
4. **Аналіз Реакцій на Події:**
   * **Дозволяє визначити, як система реагує на певні події та які дії вона виконує в різних станах.**



27. Основні компоненти та призначення діаграми компонентів. Навести приклад.

**Діаграма компонентів (Component Diagram):** Це вид діаграми UML, який використовується для візуалізації та організації компонентів системи та їх взаємодії. Компоненти можуть бути фізичними модулями програмного забезпечення, бібліотеками, файлами конфігурації, або будь-чим іншим, що представляє окрему та відділену частину системи. Основні компоненти діаграми компонентів включають:

### Основні компоненти:

1. **Компонент (Component):**
   * **Основний будівельний блок діаграми, який представляє фізичну частину системи чи логічний модуль.**
2. **Інтерфейс (Interface):**
   * **Описує методи, які може реалізувати компонент. Вказує, як інші компоненти можуть взаємодіяти з даним компонентом.**
3. **Вузол (Node):**
   * **Позначає фізичну машину, на якій працює деяка частина системи. Використовується для визначення архітектури розподіленої системи.**
4. **Залежність (Dependency):**
   * **Показує зв'язок між компонентами, де один компонент використовує або потребує інший компонент.**
5. **Зв'язок (Association):**
   * **Вказує на наявність зв'язку між компонентами. Може бути агрегацією чи композицією.**

### Призначення:

1. **Моделювання Архітектури Системи:**
   * **Діаграма компонентів дозволяє моделювати структуру та організацію компонентів у системі, а також їх взаємодію.**
2. **Аналіз та Проектування Системи:**
   * **Використовується для аналізу та проектування компонентів системи, їх інтерфейсів та взаємодії.**
3. **Визначення Модульності:**
   * **Дозволяє визначити модульні частини системи та їх границі.**
4. **Уточнення Розподіленої Архітектури:**
   * **Вказує на фізичні вузли, на яких розташовані компоненти, що допомагає уточнити розподілену архітектуру системи.**



28 Часові діаграми. Призначення та алгоритм побудови. Навести приклад з 10 етапів.

**Часові діаграми (Timing Diagrams):** Це вид діаграм UML, які використовуються для візуалізації поведінки об'єктів в системі в часі. Вони надають засоби для моделювання взаємодії між об'єктами та подій упродовж часу. Ці діаграми часто використовуються для опису сценаріїв використання, взаємодії в реальному часі та інших аспектів часової поведінки системи.

### Призначення:

1. **Візуалізація Часової Взаємодії:**
   * **Часові діаграми надають можливість візуалізувати, як об'єкти взаємодіють в реальному часі.**
2. **Моделювання Сценаріїв Використання:**
   * **Дозволяють моделювати послідовність подій та дій упродовж часу в рамках конкретного сценарію використання.**
3. **Аналіз Тривалості Операцій:**
   * **Допомагають визначити тривалість операцій та подій в системі.**

### Алгоритм Побудови:

1. **Визначення Об'єктів:**
   * **Визначте об'єкти чи компоненти, які взаємодіють у часовому контексті.**
2. **Визначення Подій:**
   * **Виокреміть ключові події та дії, які відбуваються в системі.**
3. **Визначення Тривалості:**
   * **Оцініть тривалість кожної події та дії.**
4. **Визначення Взаємодій:**
   * **Вказуйте взаємодії між об'єктами, визначаючи моменти старту та завершення кожної події.**
5. **Позначення Сигналів:**
   * **Використовуйте сигнали та стрілки для позначення взаємодії між об'єктами.**
6. **Позначення Часових Осей:**
   * **Позначте часові осі для відображення часового проміжку.**
7. **Додавання Пояснень:**
   * **Додайте пояснення до діаграми для роз'яснення значень та контексту.**
8. **Оформлення Діаграми:**
   * **Оформіть діаграму, роблячи її зрозумілою та естетичною.**



У цьому прикладі показано, як користувач та система взаємодіють під час процесу виконання замовлення в онлайн-магазині на 10 етапах від оформлення до отримання товару. Часові стрілки та позначення вказують на моменти старту та завершення кожної події.

29. Мережеві діаграми. Призначення та алгоритм побудови. Навести приклад з 10 етапів.

**Мережеві діаграми (Network Diagrams):** Це вид діаграм, які використовуються для моделювання та візуалізації послідовності та взаємодії завдань чи подій у проекті. Основне призначення - це показати логічні та фізичні зв'язки між різними елементами проекту та визначити порядок виконання робіт.

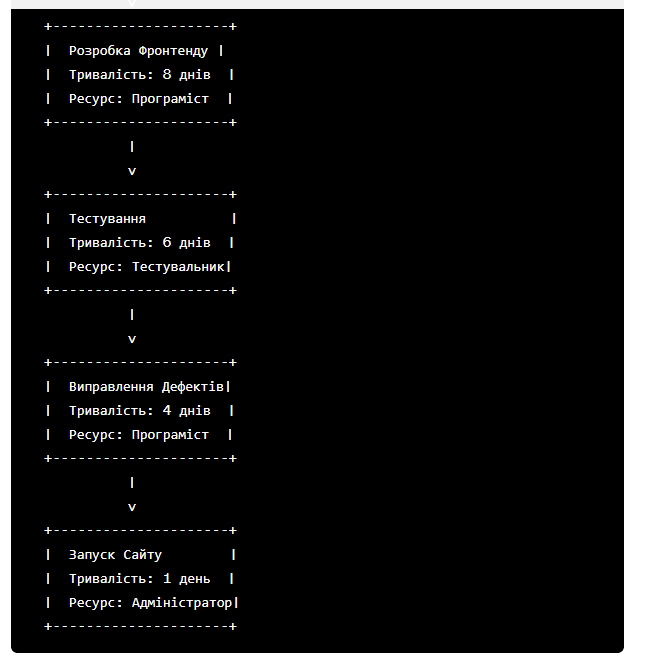
### Призначення:

1. **Моделювання Проектів:**
   * **Мережеві діаграми використовуються для моделювання структури та послідовності завдань у проекті.**
2. **Визначення Залежностей:**
   * **Показують залежності між завданнями та їх вплив на терміни виконання.**
3. **Планування Термінів:**
   * **Використовується для планування та визначення тривалості завдань та проекту в цілому.**

### Алгоритм Побудови:

1. **Визначення Завдань:**
   * **Виокреміть всі завдання, що повинні бути виконані в проекті.**
2. **Визначення Залежностей:**
   * **Визначте логічні та фізичні залежності між завданнями (яке повинно завершитися перед початком іншого).**
3. **Призначення Тривалості:**
   * **Визначте приблизний час, необхідний для виконання кожного завдання.**
4. **Створення Вершин та Ліній:**
   * **Створіть вершини для кожного завдання та лінії для показу залежностей між ними.**
5. **Визначення Критичного Шляху:**
   * **Визначте критичний шлях, який включає завдання з найбільшою тривалістю та визначає загальний термін завершення проекту.**
6. **Призначення Ресурсів:**
   * **При необхідності визначте ресурси, що використовуються для кожного завдання.**
7. **Оптимізація Термінів:**
   * **Оптимізуйте терміни завдань та проекту, вирішуючи конфлікти та використовуючи резерви часу.**
8. **Оформлення та Деталізація:**
   * **Додайте необхідну інформацію, таку як імена завдань, тривалість, терміни початку та завершення, а також ресурси.**





У цьому прикладі показано 10 етапів розробки веб-сайту, включаючи збір вимог, розробку дизайну, розробку бекенду, розробку фронтенду, тестування, виправлення дефектів та запуск сайту. Стрілки вказують на логічні залежності між завданнями, а ресурси та тривалість підкреслюють деталі проекту.

Початок форми