1. Як визначаються мови специфікацій та програмування?

    - Мови специфікацій — це формальні або напівформальні мови, які використовуються для точного та недвозначного опису вимог до програмних систем, їхніх властивостей та поведінки. Вони зосереджуються на тому, *що* повинна робити система, а не на тому, *як* вона це робить. Мови специфікацій можуть бути засновані на логіці, математичних множинах, графах або інших формальних системах.
    - Мови програмування — це формальні мови, призначені для опису алгоритмів та структур даних, які визначають, *як* програмна система повинна реалізовувати задані вимоги. Вони містять набір інструкцій, які можуть бути виконані комп'ютером для досягнення певної мети. Мови програмування відрізняються рівнем абстракції, парадигмами програмування (імперативне, об'єктно-орієнтоване, функціональне тощо) та сферою застосування.
2. Якими методами описують предметні області?
   Предметні області описують різними методами, залежно від цілей та необхідного рівня деталізації:

    - Текстовий опис: Неформальний спосіб опису у вигляді природної мови, що включає глосарій термінів, опис процесів, правил та сутностей предметної області.
    - Графічні моделі: Використання діаграм та схем для візуалізації структури та взаємозв'язків у предметній області. Приклади включають діаграми сутність-зв'язок (ERD), UML-діаграми класів, діаграми потоків даних (DFD).
    - Формальні моделі: Математичні та логічні моделі, які точно визначають сутності, їхні атрибути та відношення. Приклади включають алгебраїчні специфікації, моделі на основі теорії множин, логічні формули.
    - Прототипування: Створення ранніх версій системи або її окремих компонентів для дослідження предметної області та отримання зворотного зв'язку від зацікавлених сторін.
    - Аналіз прецедентів (Use Case Analysis): Опис взаємодії користувачів із системою для досягнення конкретних цілей, що допомагає виявити ключові функції та сутності предметної області.
    - Інтерв'ю та опитування експертів: Збір інформації безпосередньо від фахівців у предметній області для розуміння її ключових аспектів.
3. Які методи використовують для специфікації вимог до програмних систем?
   Для специфікації вимог до програмних систем використовують різноманітні

методи, що відрізняються формальністю та рівнем деталізації:

- Неформальні специфікації: Опис вимог природною мовою, часто у вигляді документів, що включають текстові описи, сценарії використання та загальні цілі системи.
- Напівформальні специфікації: Використання комбінації природної мови та формальних елементів, таких як діаграми (UML, BPMN), таблиці рішень, структурований текст.
- Формальні специфікації: Використання математичних та логічних формалізмів для точного та недвозначного опису вимог. Приклади включають специфікації на основі логіки першого порядку, теорії множин (Z-метод), алгебраїчних специфікацій, темпоральних логік (LTL, CTL).
- Специфікації на основі моделей: Створення абстрактних моделей поведінки системи (наприклад, автомати зі скінченною кількістю станів, мережі Петрі) для опису вимог до динаміки системи.
- Специфікації на основі прецедентів (Use Case Specifications): Детальний опис взаємодії користувачів із системою для кожного випадку використання, включаючи основний потік подій та альтернативні сценарії.
- Специфікації, керовані поведінкою (Behavior-Driven Development - BDD): Опис вимог у вигляді сценаріїв, написаних природною мовою, які можуть бути автоматизовані для тестування.

4. Які засади B-методу розробки програм?
   B-метод — це формальний метод розробки програмного забезпечення, заснований на математичній теорії множин та логіці першого порядку. Його ключові засади включають:

   - Формальна специфікація: Використання абстрактних машин для формального опису вимог до системи. Абстрактна машина визначає стан системи та операції, які можуть змінювати цей стан.
   - Поступове уточнення (Refinement): Розробка відбувається шляхом послідовного уточнення абстрактної специфікації до конкретної реалізації. Кожен крок уточнення зберігає властивості попереднього рівня.
   - Доведення коректності: На кожному кроці уточнення формально доводиться, що нижчий рівень специфікації є правильною реалізацією вищого рівня. Це забезпечує коректність розробленого програмного забезпечення.

- Використання інструментів: Існують спеціалізовані інструменти (наприклад, Atelier B), які підтримують процес формальної специфікації, доведення та генерації коду.
- Акцент на безпеці та надійності: B-метод особливо корисний для розробки критично важливих систем, де висока надійність та безпека є пріоритетними.

5. Які засади TLA-методу розробки програм?
   TLA (Temporal Logic of Actions) — це формальний метод специфікації та верифікації паралельних та розподілених систем, розроблений Леслі Лемпортом. Його основні засади включають:

   - Специфікація поведінки: Використання темпоральної логіки дій для опису можливих послідовностей станів та переходів між ними. Специфікація описує, *що* система робить, а не *як* вона це робить.
   - Абстракція: Можливість створення специфікацій на різних рівнях абстракції, що дозволяє зосередитися на ключових властивостях системи на ранніх етапах розробки.
   - Верифікація за допомогою модельної перевірки: Використання інструменту TLC (TLA+ Model Checker) для автоматичної перевірки властивостей специфікації шляхом дослідження всіх можливих станів та переходів.
   - Доведення теорем: Можливість формального доведення властивостей специфікації за допомогою математичних міркувань.
   - Акцент на паралелізмі та розподіленості: TLA+ особливо добре підходить для специфікації та верифікації складних паралельних та розподілених алгоритмів та систем.
6. Які засади RAISE-методу розробки програм?
   RAISE (Rigorous Approach to Industrial Software Engineering) — це формальний метод розробки програмного забезпечення, який поєднує алгебраїчні специфікації та модельовані об'єкти. Його ключові засади включають:

   - Багаторівнева специфікація: Розробка починається з абстрактних специфікацій, які поступово уточнюються до конкретних реалізацій.
   - Використання алгебраїчних специфікацій: Опис типів даних та операцій над ними за допомогою алгебраїчних рівнянь.
   - Використання модельованих об'єктів: Опис поведінки системи за допомогою моделей, що відображають її стан та переходи між станами.

- Підтримка різних стилів специфікації: Можливість використання як імпліцитних (опис бажаних властивостей), так і експліцитних (опис алгоритмів) специфікацій.
- Формальне обґрунтування: На кожному кроці уточнення необхідно формально довести, що нижчий рівень специфікації відповідає вищому рівню.
- Інтегрований підхід: RAISE надає комплексний підхід до розробки, що охоплює специфікацію, проектування та верифікацію.

7. Які засади Z-методу розробки програм?
   Z-метод — це формальний метод специфікації, заснований на теорії множин та логіці першого порядку. Його основні засади включають:

   - Використання формальної мови Z: Специфікації пишуться формальною мовою Z, яка включає математичні нотації для множин, відношень, функцій та логічних предикатів.
   - Специфікація станів та операцій: Система моделюється як набір станів, які визначаються за допомогою схем станів (state schemas), та операцій, які змінюють ці стани, що описуються за допомогою схем операцій (operation schemas).
   - Передумови та післяумови: Операції специфікуються за допомогою передумов (умов, які повинні бути істинними перед виконанням операції) та післяумов (умов, які повинні бути істинними після успішного завершення операції).
   - Акцент на точності та недвозначності: Формальний характер Z-специфікацій дозволяє уникнути неоднозначностей, властивих неформальним описам.
   - Основа для верифікації: Z-специфікації можуть бути використані для формального доведення властивостей системи та коректності її реалізації.
8. Які логічні формалізми використовують для специфікацій програм?
   Для специфікацій програм використовують різноманітні логічні формалізми, залежно від типу властивостей, які необхідно описати:

   - Логіка першого порядку (First-Order Logic - FOL): Використовується для опису статичних властивостей даних та відношень між ними. Багато формальних методів (наприклад, B-метод, Z-метод) базуються на FOL.
   - Темпоральні логіки (Temporal Logics): Використовуються для опису поведінки систем у часі. Приклади включають лінійну темпоральну логіку (Linear Temporal Logic - LTL) та логіку обчислювальних дерев

(Computation Tree Logic - CTL). Вони дозволяють специфікувати властивості, що стосуються послідовності подій та станів.
    - Модальні логіки (Modal Logics): Дозволяють описувати можливості та необхідності, а також знання та переконання агентів у системах. Можуть використовуватися для специфікації властивостей безпеки та живості в паралельних системах.
    - Логіка Гоара (Hoare Logic): Аксіоматична система для формальної верифікації програм. Вона використовує трійки Гоара {P}C{Q}, де P — передумова, C — команда, а Q — післяумова.
    - Динамічна логіка (Dynamic Logic): Розширення логіки першого порядку, що включає модальні оператори, пов'язані з виконанням програм. Дозволяє специфікувати властивості, що стосуються результатів виконання програм.
9. Як використовується класична та некласична логіка для специфікацій програм?

    - Класична логіка (Classical Logic): Зазвичай включає логіку висловлювань та логіку першого порядку. Вона є основою для багатьох формальних методів специфікації, таких як Z-метод та B-метод. Класична логіка використовується для опису інваріантів, передумов, післяумов та інших статичних властивостей програм. Її строгі правила виведення дозволяють формально доводити властивості специфікацій.
    - Некласична логіка (Non-Classical Logic): Включає логіки, які відрізняються від класичної логіки в своїх аксіомах або правилах виведення. Для специфікацій програм можуть використовуватися різні види некласичної логіки:
        - Темпоральна логіка: Для специфікації поведінки систем у часі (наприклад, "завжди після запиту йде відповідь").
        - Модальна логіка: Для специфікації можливостей, необхідностей та знань (наприклад, "можливо, система досягне стану помилки").
        - Інтуїціоністська логіка: Може використовуватися в контексті конструктивних доведень програм.
        - Нечітка логіка (Fuzzy Logic): Для специфікації систем, що працюють з нечіткими або неповними даними.
10. Як визначаються темпоральні та модальні логіки?

- Темпоральні логіки (Temporal Logics): Це розширення класичної логіки, які додають оператори для міркування про час та послідовність подій. Основні темпоральні оператори включають:
  - G (Globally): Властивість є істинною у всіх майбутніх станах.
  - F (Finally): Властивість стане істинною в якомусь майбутньому стані (або вже є істинною).
  - X (Next): Властивість буде істинною в наступному стані.
  - U (Until): Властивість P залишається істинною до тих пір, поки не стане істинною властивість Q. Існують різні види темпоральних логік, такі як LTL (лінійна темпоральна логіка, де розглядається єдина послідовність станів) та CTL (логіка обчислювальних дерев, де розглядаються всі можливі майбутні шляхи виконання).
- Модальні логіки (Modal Logics): Це розширення класичної логіки, які додають модальні оператори, що виражають необхідність та можливість. Основні модальні оператори включають:
  - □ (Box, Necessity): Властивість є істинною у всіх доступних світах (або станах).
  - ◊ (Diamond, Possibility): Властивість є істинною хоча б в одному доступному світі (або стані). Значення "доступності" залежить від конкретної модальної логіки (наприклад, епістемічна логіка для знання, де "доступні світи" — це світи, сумісні з тим, що агент знає). У контексті специфікацій програм, модальні логіки можуть використовуватися для міркування про безпеку (наприклад, "ніколи не досягається небажаний стан") та живучість (наприклад, "завжди зрештою відбудеться певна подія").

11. Як визначаються аксіоматичні методи специфікації програм?
    Аксіоматичні методи специфікації програм визначають значення програмних конструкцій за допомогою набору логічних правил (аксіом та правил виведення). Замість опису процесу виконання програми (як в операційній семантиці) або визначення математичних об'єктів, що представляють програму (як в денотаційній семантиці), аксіоматичні методи встановлюють формальні зв'язки між станами програми до та після виконання її фрагментів.

    Основна ідея полягає у визначенні правил, які дозволяють виводити твердження про стан програми після виконання певної команди, виходячи з тверджень про стан програми до її виконання. Найвідомішим прикладом аксіоматичного методу є логіка Гоара.

Аксіоматичні методи зазвичай включають:

- Аксіоми для базових команд: Визначають, як змінюється стан програми після виконання простих команд, таких як присвоєння.
- Правила виведення для складених конструкцій: Описують, як виводити твердження про виконання складених команд (послідовність, умовні оператори, цикли) на основі тверджень про їхні складові частини.

12. Як визначається логіка Флойда-Хоара та які властивості вона має?
    Логіка Флойда-Хоара (часто просто згадується як логіка Гоара) — це аксіоматична система для формальної верифікації програм, зокрема для доведення їхньої часткової коректності. Вона базується на понятті трійки Гоара (Hoare triple):
    {P}C{Q}
    де:

    - P — це передумова (precondition), логічне твердження про стан програми перед виконанням команди C.
    - C — це команда (statement) програми.
    - Q — це післяумова (postcondition), логічне твердження про стан програми після виконання команди C, якщо виконання завершилося.
13. Логіка Флойда-Хоара визначає набір аксіом для базових команд (наприклад, присвоєння, пропуск) та правил виведення для складених конструкцій (послідовність, умовні оператори, цикли). Ці правила дозволяють виводити нові трійки Гоара на основі вже відомих.

    Властивості логіки Флойда-Хоара:

    - Звучність (Soundness): Якщо трійку Гоара {P}C{Q} можна довести за допомогою правил логіки Флойда-Хоара, то вона є істинною. Це означає, що якщо передумова P виконується перед виконанням команди C, і виконання C завершується, то після виконання буде істинною післяумова Q. Звучність гарантує, що доведені властивості дійсно виконуються.
    - Часткова коректність: Логіка Флойда-Хоара в своїй базовій формі доводить лише часткову коректність. Це означає, що якщо програма завершує виконання, то результат буде відповідати специфікації

(післяумові). Вона не гарантує, що програма обов'язково завершиться.
    - Виразність (Expressiveness): Логіка предикатів, яка використовується для вираження передумов та післяумов, повинна бути достатньо виразною для опису станів програми та бажаних властивостей.
14. Повнота логіки Флойда-Хоара.
    Повнота логіки Флойда-Хоара є складним питанням і залежить від різних факторів, включаючи виразність логіки предикатів, що використовується для опису передумов та післяумов, а також від особливостей мови програмування.

    У загальному випадку, логіка Флойда-Хоара не є повною відносно всіх можливих програм та їхніх властивостей. Це означає, що існують істинні трійки Гоара {P}C{Q}, які не можуть бути доведені за допомогою правил логіки Флойда-Хоара.

    Однак, для певних класів програм і за певних обмежень на логіку предикатів, можуть існувати результати щодо відносної повноти. Наприклад, якщо логіка предикатів є достатньо виразною (наприклад, включає арифметику першого порядку) і мова програмування не містить складних особливостей (таких як довільні переходи `goto`), то можна досягти відносної повноти. Відносна повнота означає, що якщо істинна трійка Гоара існує, то її доведення може бути зведене до доведення певних тверджень у логіці предикатів.

    Проблема повноти частково пов'язана з неможливістю ефективно виразити всі можливі інваріанти циклів у логіці першого порядку.

15. Які особливості має семантико-синтаксична технологія розробки програм?
    Семантико-синтаксична технологія розробки програм (часто згадується як Semantic-Syntactic Technology - SST) — це підхід до розробки програмного забезпечення, який інтегрує синтаксичний аналіз та семантичне розуміння на різних етапах життєвого циклу розробки. Її особливості включають:

    - Орієнтація на формальне представлення знань: SST передбачає використання формальних моделей та представлень для опису предметної області, вимог та самої програми. Це може включати онтології, семантичні мережі, формальні граматики.
    - Інтеграція синтаксису та семантики: На відміну від традиційних підходів, де синтаксичний та семантичний аналіз часто розглядаються

окремо, SST намагається поєднати їх для більш глибокого розуміння програмного коду та вимог.
- Автоматизація процесів розробки: Використання формальних моделей та семантичного аналізу дозволяє автоматизувати багато рутинних завдань, таких як генерація коду, верифікація, документування та тестування.
- Підтримка розуміння та супроводження коду: Семантична інформація, пов'язана з програмним кодом, полегшує його розуміння, аналіз залежностей та внесення змін під час супроводження.
- Покращення якості програмного забезпечення: Завдяки формалізації та автоматизованій перевірці, SST може сприяти створенню більш надійного та коректного програмного забезпечення.
- Використання метаданих: SST активно використовує метадані для опису синтаксичної структури та семантичного значення елементів програми та предметної області.

16. Які особливості має метод послідовних уточнень?
    Метод послідовних уточнень (Stepwise Refinement) — це стратегія проектування програмного забезпечення, яка полягає у поступовому переході від абстрактного опису розв'язку задачі до його детальної реалізації. Його основні особливості включають:

    - Розподіл складності: Замість одразу намагатися розв'язати всю складну задачу, її розбивають на менші, більш керовані підзадачі.
    - Абстракція на початкових етапах: Розробка починається з опису розв'язку на високому рівні абстракції, без зайвих деталей реалізації.
    - Поступова деталізація: На кожному наступному кроці абстрактні описи уточнюються, додаються деталі алгоритмів та структур даних.
    - Збереження коректності на кожному кроці: Метод передбачає, що кожне уточнення повинно зберігати функціональність та коректність попереднього рівня.
    - Ієрархічна структура: Процес уточнень природно призводить до ієрархічної структури програми, де складні операції розбиваються на послідовність простіших.
    - Покращення розуміння та проектування: Поступовий підхід полегшує розуміння розв'язку та прийняття обґрунтованих проектних рішень на кожному етапі.
    - Зменшення кількості помилок: Розбиття задачі на менші частини та контрольоване додавання деталей допомагає зменшити ймовірність внесення помилок.

17. Які особливості у розробку програм вносять об’єктно-орієнтовані методи?
    Об'єктно-орієнтовані методи (Object-Oriented Methods) вносять ряд суттєвих особливостей у процес розробки програмного забезпечення:

    - Абстракція: Дозволяє зосереджуватися на важливих аспектах об'єктів, ігноруючи несуттєві деталі. Класи інкапсулюють дані (атрибути) та поведінку (методи), надаючи абстрактний інтерфейс для взаємодії.
    - Інкапсуляція: Приховує внутрішню реалізацію об'єкта від зовнішнього світу, забезпечуючи захист даних та можливість змінювати реалізацію без впливу на клієнтський код.
    - Успадкування: Дозволяє створювати нові класи на основі вже існуючих, повторно використовуючи їхню функціональність та додаючи або змінюючи поведінку. Це сприяє зменшенню дублювання коду та покращує організацію ієрархій класів.
    - Поліморфізм: Дозволяє об'єктам різних класів реагувати на одні й ті самі повідомлення (виклики методів) по-різному, залежно від їхнього типу. Це підвищує гнучкість та розширюваність коду.
    - Моделювання реального світу: Об'єктно-орієнтований підхід полегшує моделювання сутностей та взаємодій з реального світу у програмному коді, що робить його більш інтуїтивно зрозумілим.
    - Модульність: Програми розбиваються на незалежні об'єкти, що полегшує розробку, тестування та супроводження окремих частин системи.
    - Повторне використання коду: Механізми успадкування та композиції сприяють повторному використанню вже розробленого коду, що зменшує час та вартість розробки.
18. Яка мета стандартів програмування?
    Мета стандартів програмування полягає у встановленні набору правил, рекомендацій та найкращих практик, яких повинні дотримуватися розробники програмного забезпечення. Основні цілі стандартів програмування включають:

    - Підвищення якості коду: Стандарти сприяють написанню більш читабельного, зрозумілого, надійного та підтримуваного коду.
    - Забезпечення консистентності: Дотримання стандартів робить код більш однорідним, що полегшує його розуміння та спільну роботу над проектом.
    - Покращення супроводження: Консистентний та добре структурований код легше модифікувати, виправляти помилки та розширювати в майбутньому.

- Спрощення командної розробки: Стандарти забезпечують загальні правила для всіх членів команди, що полегшує інтеграцію різних частин проекту.
- Зменшення кількості помилок: Дотримання перевірених практик може допомогти уникнути типових помилок програмування.
- Покращення переносимості коду: Стандарти можуть включати рекомендації щодо написання платформно-незалежного коду.
- Полегшення навчання нових розробників: Чіткі стандарти допомагають новим членам команди швидше освоїтися в проекті.

19. Як використовують технологічні та інструментальні засоби специфікації та розробки програм?
    Технологічні та інструментальні засоби відіграють ключову роль у процесах специфікації та розробки програмного забезпечення, забезпечуючи підтримку різних етапів життєвого циклу:

    - Засоби для специфікації вимог:
        - Текстові редактори та системи керування документацією: Для створення та організації неформальних специфікацій (наприклад, MS Word, Confluence).
        - Інструменти для моделювання UML та BPMN: Для створення напівформальних специфікацій у вигляді діаграм (наприклад, Enterprise Architect, draw.io).
        - Інструменти для формальних специфікацій: Підтримка формальних мов та методів (наприклад, Atelier B для B-методу, TLC для TLA+, Isabelle/HOL для формальних доведень).
        - Інструменти для керування вимогами (Requirements Management Tools): Для збору, документування, відстеження та керування вимогами (наприклад, Jira, Confluence з плагінами, Polarion).
        - Інструменти для BDD (Behavior-Driven Development): Для написання та автоматизації специфікацій у вигляді сценаріїв (наприклад, Cucumber, SpecFlow).
    - Середовища інтегрованої розробки (Integrated Development Environments - IDE): Надають комплексне середовище для написання, редагування, компіляції, налагодження та тестування коду (наприклад, IntelliJ IDEA, Visual Studio, Eclipse).
    - Системи контролю версій (Version Control Systems - VCS): Для керування змінами у коді, спільної роботи та відстеження історії проекту (наприклад, Git, SVN).

- Інструменти для автоматизації збірки (Build Automation Tools): Для автоматизації процесів компіляції, тестування та пакування програмного забезпечення (наприклад, Maven, Gradle, Jenkins).
- Інструменти для тестування: Різноманітні засоби для автоматизованого тестування (юніт-тестування, інтеграційне тестування, системне тестування, acceptance testing) (наприклад, JUnit, TestNG, Selenium, Cypress).
- Інструменти для аналізу коду: Статичні та динамічні аналізатори коду для виявлення потенційних помилок, вразливостей та проблем з якістю коду (наприклад, SonarQube, Checkstyle, FindBugs).
- Інструменти для розгортання (Deployment Tools): Для автоматизації процесу розгортання програмного забезпечення на цільових середовищах (наприклад, Docker, Kubernetes, Ansible).