1. **Какие фреймворки используются для модульного тестирования в Java?**
2. **Что такое JUnit и какие преимущества он предлагает для тестирования Java приложений?**
3. **Какие основные компоненты JUnit и как они работают вместе?**

---

1. Какие фреймворки используются для модульного тестирования в Java?

Основные:

1. **JUnit**
   - Стандарт для модульного тестирования Java.
   - Поддерживает аннотации, assertions, lifecycle методов и интеграцию с CI/CD.
2. **TestNG**
   - Альтернатива JUnit.
   - Более гибкий контроль порядка выполнения тестов, поддержка параметризованных и параллельных тестов.
3. **Mockito**
   - Фреймворк для **мокирования** зависимостей (mock objects).
   - Используется вместе с JUnit или TestNG.
4. **PowerMock**
   - Расширение Mockito для мокирования статических методов, final классов.
5. **AssertJ / Hamcrest**
   - Библиотеки для удобного **написания выражений проверки** (assertions).
6. **Spring Test / Spring Boot Test**
   - Для интеграционного тестирования Spring приложений.
   - Поддержка контекста Spring, моков, web environment.

---

2. Что такое JUnit и какие преимущества он предлагает?

JUnit — это **фреймворк для модульного тестирования Java-программ**, который позволяет:

- Создавать, запускать и организовывать тесты.
- Проверять корректность работы отдельных методов или классов.

Преимущества JUnit:

1. **Стандартизированная структура тестов**
   - Аннотации для определения методов тестирования и lifecycle: `@Test`, `@BeforeEach`, `@AfterEach` и т.д.
2. **Автоматизация тестирования**
   - Тесты можно запускать через IDE, build tools (Maven, Gradle), CI/CD.
3. **Поддержка Assertions**
   - Проверка ожидаемых значений через `assertEquals`, `assertTrue`, `assertThrows` и др.
4. **Управление зависимостями и lifecycle**
   - `@BeforeAll`, `@BeforeEach`, `@AfterEach`, `@AfterAll` позволяют настраивать окружение и чистить его после тестов.
5. **Интеграция с другими инструментами**
   - Mockito, Spring Test, Jacoco (для покрытия кода), CI/CD системы.

---

3. Основные компоненты JUnit и как они работают вместе

Компонент	Описание	Пример использования
Test Class	Класс, содержащий тесты	`public class UserServiceTest { ... }`
Test Method	Метод, аннотированный `@Test`	`@Test void testCreateUser() { ... }`
Assertions	Проверки, которые подтверждают корректность работы	`assertEquals(expected, actual)`
Lifecycle Annotations	Методы для подготовки и очистки тестового окружения	`@BeforeEach void setup() { ... }`
Test Suite	Группа тестов, объединённых в набор	Можно запускать весь модуль или пакет тестов
Exception Testing	Проверка выбрасываемых исключений	`assertThrows(IllegalArgumentException.class, () -> method())`
Parameterized Tests	Позволяет запускать тест с разными входными данными	`@ParameterizedTest` + `@ValueSource`

Как они взаимодействуют

1. **JUnit запускает тестовый класс**.
2. Перед каждым тестом вызывается `@BeforeEach` (инициализация).
3. Выполняется метод с `@Test`.
4. Проверки выполняются через **Assertions**.
5. После теста вызывается `@AfterEach` (очистка).
6. При необходимости можно использовать `@BeforeAll` и `@AfterAll` для работы с ресурсами один раз на весь класс.

1. **Как работает процесс написания тестов с использованием JUnit?**
2. **Опишите процесс настройки и запуска тестов с использованием JUnit и какие основные аннотации используются в этом процессе.**
3. **Какие существуют виды тестирования ПО и для каких целей они используются?**

---

1. Как работает процесс написания тестов с использованием JUnit?

Процесс можно разделить на несколько шагов:

1. **Создание тестового класса**
   - Для каждого тестируемого класса создаётся отдельный тестовый класс, обычно с суффиксом `Test`.
   - Пример:
   - `public class UserServiceTest { ... }`
2. **Определение тестовых методов**
   - Каждый метод, проверяющий отдельную функциональность, помечается аннотацией `@Test`.
   - Внутри методов используются **assertions** для проверки результата.
3. **Подготовка тестового окружения**
   - Инициализация объектов и зависимостей через `@BeforeEach` или `@BeforeAll`.
4. **Выполнение теста**
   - JUnit запускает тестовый метод.
   - Выполняются assertions, проверяющие правильность результата.
5. **Очистка ресурсов**
   - После выполнения теста вызывается `@AfterEach` для освобождения ресурсов.
   - Для глобальных ресурсов — `@AfterAll`.
6. **Анализ результатов**
   - Тест считается успешным, если **все assertions прошли**.
   - Ошибки или исключения отмечаются как **failed**.

---

2. Процесс настройки и запуска тестов в JUnit и основные аннотации

Настройка

1. **Добавление зависимости**
   - Через Maven:
   - `<dependency>`
   - `  <groupId>org.junit.jupiter</groupId>`
   - `  <artifactId>junit-jupiter</artifactId>`
   - `  <version>5.10.0</version>`
   - `  <scope>test</scope>`
   - `</dependency>`
   - Через Gradle:
   - `testImplementation 'org.junit.jupiter:junit-jupiter:5.10.0'`
2. **Создание тестовых классов и методов**
   - Организация структуры пакетов для удобства запуска и поддержки.
3. **Выбор среды запуска**
   - IDE (IntelliJ, Eclipse)
   - Build Tools (Maven, Gradle)
   - CI/CD (Jenkins, GitLab CI, GitHub Actions)

Основные аннотации JUnit

Аннотация	Назначение
`@Test`	Обозначает метод теста
`@BeforeEach`	Выполняется **перед каждым тестом** (инициализация объектов)
`@AfterEach`	Выполняется **после каждого теста** (очистка ресурсов)
`@BeforeAll`	Выполняется один раз **до всех тестов класса**
`@AfterAll`	Выполняется один раз **после всех тестов класса**
`@Disabled`	Игнорирует тест (например, временно)
`@ParameterizedTest`	Позволяет запускать один тест с разными входными данными
`@ValueSource`, `@CsvSource`, `@MethodSource`	Поставщики данных для параметризованных тестов
`assertThrows()`	Проверка выбрасываемого исключения

---

3. Виды тестирования ПО и их цели

Вид тестирования	Цель	Примеры
Модульное (Unit Testing)	Проверка отдельного метода или класса	JUnit, Mockito
Интеграционное (Integration Testing)	Проверка взаимодействия нескольких компонентов	Spring Boot Test, TestContainers
Системное (System Testing)	Проверка всей системы в целом	Selenium, JMeter
Приёмочное (Acceptance Testing)	Проверка соответствия требований	Cucumber, FitNesse
Регрессионное (Regression Testing)	Проверка, что новые изменения не сломали старую функциональность	Автотесты в CI/CD
Нагрузочное/Performance Testing	Проверка работы под высокой нагрузкой	JMeter, Gatling
Тестирование безопасности (Security Testing)	Проверка уязвимостей	OWASP ZAP, Burp Suite
Smoke Testing	Быстрая проверка основных функций после билда	Набор базовых тестов
Exploratory / Ad-hoc	Ручное исследовательское тестирование	Тестировщик вручную проверяет UI и функционал

1. **Какие стратегии можно применить для тестирования микросервисов?**
2. **Как реализовать параметризованные тесты в JUnit 5?**
3. **Какие особенности тестирования реактивных приложений и какие инструменты для этого используются?**

---

1. Стратегии тестирования микросервисов

Тестирование микросервисов требует комбинации подходов:

1. Unit Testing (модульное тестирование)

- Тестирует отдельный сервис или класс.
- Используются **JUnit + Mockito** для мокирования зависимостей.

2. Integration Testing (интеграционное тестирование)

- Проверка взаимодействия сервисов с внешними компонентами: БД, очередями, API.
- Используются: **Spring Boot Test**, **TestContainers** (Docker), **Embedded DB**.

3. Contract Testing (тестирование контрактов)

- Проверяет соответствие API между сервисами.
- Инструменты: **Pact**, **Spring Cloud Contract**.
- Позволяет гарантировать, что изменения в одном сервисе не сломают другой.

4. End-to-End Testing (E2E)

- Полная проверка сценариев работы системы.
- Инструменты: **Selenium**, **Cypress**, **Postman/Newman**.

5. Consumer-Driven Testing

- Тестирование сервиса на основе ожиданий потребителей.
- Обычно для API, чтобы обеспечить согласованность контрактов.

---

2. Параметризованные тесты в JUnit 5

JUnit 5 позволяет запускать один тест с разными входными данными.

Пример с `@ParameterizedTest`

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;
import static org.junit.jupiter.api.Assertions.assertTrue;

class MathTest {

    @ParameterizedTest
    @ValueSource(ints = {2, 4, 6, 8})
    void testEvenNumbers(int number) {
        assertTrue(number % 2 == 0);
    }
}
```

Другие поставщики данных

Поставщик	Описание
`@ValueSource`	Массив простых значений (int, String, etc.)
`@CsvSource`	Несколько значений в формате CSV
`@CsvFileSource`	Данные из CSV файла
`@MethodSource`	Метод, возвращающий поток значений
`@EnumSource`	Перечисления (enum)

Особенности:

- Один тестовый метод выполняется несколько раз с разными входными данными.
- Полезно для проверки логики с множеством вариантов.

---

3. Особенности тестирования реактивных приложений

Реактивные приложения используют **асинхронные потоки данных** (`Mono`, `Flux` в Spring WebFlux, Project Reactor).

Основные вызовы

1. Асинхронность → тесты должны учитывать **ожидание завершения потоков**.
2. Потоки могут быть **бесконечными** → необходимо контролировать таймауты.
3. Обработка ошибок → нужно проверять сигналы ошибок (`onError`).

Инструменты

1. **StepVerifier (Project Reactor)**
   - Позволяет тестировать реактивные потоки последовательно:

```
StepVerifier.create(Flux.just(1,2,3))
            .expectNext(1)
            .expectNext(2)
            .expectNext(3)
            .verifyComplete();
```

7. **WebTestClient (Spring WebFlux)**
   - Для тестирования реактивных REST API:

```
webTestClient.get().uri("/users")
             .exchange()
               .expectStatus().isOk()
               .expectBodyList(User.class).hasSize(3);
```

12. **Testcontainers** и **Embedded DB** для интеграции с БД в реактивных приложениях.

Особенности

- Не использовать `Thread.sleep()` для ожидания результатов.
- Использовать StepVerifier или подобные инструменты для контроля последовательности сигналов.
- Контролировать backpressure и таймауты.

---

1. **Что такое заглушка (stub) в контексте модульного тестирования?**
2. **Какова основная цель использования заглушек в тестировании?**
3. **В чем разница между заглушками и макетами (mocks)?**

---

1. Что такое заглушка (stub) в контексте модульного тестирования?

Stub (заглушка) — это **искусственная реализация зависимости**, используемая в тесте вместо реального объекта.

- Обычно возвращает **предопределённые данные**.
- Не содержит сложной логики.
- Применяется для изоляции тестируемого компонента от внешних зависимостей (БД, веб-сервисы, API).

Пример:

```
class UserServiceTest {

    private UserRepository stubRepository = new UserRepository() {
        @Override
        public User findById(int id) {
            return new User(id, "Test User"); // возвращаем фиксированные
данные
        }
    };

    @Test
    void testGetUserName() {
        UserService userService = new UserService(stubRepository);
        String name = userService.getUserName(1);
        assertEquals("Test User", name);
    }
}
```

- Здесь `stubRepository` заменяет реальную БД и возвращает заранее заданный объект.

---

2. Основная цель использования заглушек

1. **Изоляция тестируемого кода**
   - Тестируем только логику класса, без влияния внешних сервисов.
2. **Ускорение тестов**
   - Нет необходимости обращаться к реальным БД или API → тесты выполняются быстро.
3. **Предсказуемость и стабильность**
   - Результаты теста не зависят от внешнего окружения.
4. **Упрощение настройки сложных зависимостей**
   - Легко имитировать редкие сценарии (например, ошибки сервиса, пустой ответ).

---

3. Разница между заглушками (stubs) и макетами (mocks)

Характеристика	Stub (Заглушка)	Mock (Макет)
Цель	Возврат фиксированных данных для теста	Проверка взаимодействий с объектом (вызовы методов, параметры)
Содержит логику?	Обычно нет, только заранее заданные ответы	Может проверять вызовы и состояния
Применение	Когда важен результат метода	Когда важны **вызовы методов и порядок взаимодействия**
Пример использования	Заглушка репозитория возвращает объект	Проверка, что метод `save()` был вызван ровно один раз с нужным объектом
Инструменты	Можно вручную или с помощью Mockito	Mockito, EasyMock, JMock

Пример с Mockito для mock:

```
UserRepository mockRepository = Mockito.mock(UserRepository.class);
User user = new User(1, "Test");
Mockito.when(mockRepository.findById(1)).thenReturn(user);

UserService service = new UserService(mockRepository);
service.getUserName(1);

// Проверяем, что метод был вызван
Mockito.verify(mockRepository, Mockito.times(1)).findById(1);
```

- **Stub** → просто возвращает данные.
- **Mock** → проверяет взаимодействие и может возвращать данные.

---

1. **Какие есть типы заглушек и в чем их основные отличия?**
2. **Как заглушка может быть использована для имитации исключений в тестах?**
3. **В каких случаях предпочтительнее использовать заглушки, а не реальные объекты в тестировании?**

---

1. Типы заглушек и их основные отличия

В тестировании часто выделяют несколько типов заглушек:

Тип заглушки	Описание	Пример
Fixed Value Stub	Возвращает заранее заданные данные без логики	Метод `findUserById()` всегда возвращает объект `User(1, "Test")`
Parameterized Stub	Возвращает разные данные в зависимости от входных параметров	Если `id=1` → User1, если `id=2` → User2
State-Based Stub	Поддерживает состояние, влияет на последующие вызовы	Метод `getNextOrder()` возвращает разный объект при каждом вызове
Exception Stub	Искусственно выбрасывает исключение	Метод `findUserById(-1)` бросает `IllegalArgumentException`

Основное отличие — **уровень логики и реакция на входные данные**. Fixed Value — самый простой, Exception Stub — имитирует ошибки.

---

2. Использование заглушки для имитации исключений

Заглушка может помочь протестировать обработку ошибок в коде:

Пример с JUnit и Mockito:

```
UserRepository stubRepository = Mockito.mock(UserRepository.class);

// Настраиваем заглушку на выброс исключения
Mockito.when(stubRepository.findById(-1))
       .thenThrow(new IllegalArgumentException("Invalid ID"));

UserService service = new UserService(stubRepository);

assertThrows(IllegalArgumentException.class, () -> service.getUserName(-1));
```

- Полезно, когда нужно проверить, как сервис реагирует на ошибки БД, внешних API или недопустимые входные данные.

3. Когда использовать заглушки вместо реальных объектов

Предпочтительно использовать заглушки, когда:

1. **Внешние зависимости недоступны или нестабильны**
   - БД, веб-сервисы, микросервисы.
2. **Требуется ускорить выполнение тестов**
   - Заглушки работают мгновенно, без сетевых запросов.
3. **Нужно протестировать редкие или опасные сценарии**
   - Исключения, ошибки API, сетевые сбои.
4. **Необходимо изолировать тестируемый модуль**
   - Проверка **только бизнес-логики**, без влияния зависимостей.
5. **Сложность создания реальных объектов слишком высокая**
   - Конфигурация БД, внешние сервисы, ресурсоёмкие объекты.

Ключевая идея:

- Заглушка → безопасная имитация поведения внешней зависимости.
- Реальный объект → используется только для интеграционных или end-to-end тестов.

1. **Как реализовать параметризированную заглушку с использованием фреймворка для тестирования на Java?**
2. **Можете ли вы привести пример использования заглушек для тестирования взаимодействия между слоями приложения?**
3. **Какие принципы следует учитывать при проектировании заглушек для обеспечения их эффективного использования?**

---

1. Параметризированная заглушка в Java

Параметризированная заглушка — это заглушка, которая возвращает **разные результаты в зависимости от входных данных**.

Пример с Mockito и JUnit 5

```
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import static org.junit.jupiter.api.Assertions.*;

class UserServiceTest {

    @Test
    void testParameterizedStub() {
        UserRepository stubRepository = Mockito.mock(UserRepository.class);

        // Настройка поведения заглушки в зависимости от параметра
        Mockito.when(stubRepository.findById(1))
               .thenReturn(new User(1, "Alice"));
        Mockito.when(stubRepository.findById(2))
               .thenReturn(new User(2, "Bob"));

        UserService service = new UserService(stubRepository);

        assertEquals("Alice", service.getUserName(1));
        assertEquals("Bob", service.getUserName(2));
    }
}
```

Особенности:

- Можно имитировать сложные сценарии без обращения к реальной БД или API.
- Полезно для проверки разных входных значений и ветвлений логики.

---

2. Использование заглушек для тестирования взаимодействия между слоями

Заглушки часто применяются для **изоляции слоев** (например, сервисного и репозитория):

Пример

```
class OrderServiceTest {

    @Test
    void testCreateOrder() {
        // Заглушка репозитория
        OrderRepository stubRepo = new OrderRepository() {
            @Override
            public void save(Order order) {
                // ничего не сохраняем, просто имитируем метод
            }
        };

        PaymentService stubPayment = new PaymentService() {
            @Override
            public boolean processPayment(double amount) {
                return true; // имитация успешной оплаты
            }
        };

        OrderService orderService = new OrderService(stubRepo, stubPayment);

        boolean result = orderService.createOrder(new Order(100));
        assertTrue(result); // проверяем бизнес-логику без реальных
зависимостей
    }
}
```

Преимущества:

- Можно проверить бизнес-логику сервиса без доступа к БД или внешнему API.
- Упрощает написание тестов и ускоряет их выполнение.

---

3. Принципы проектирования заглушек для эффективного использования

1. **Минимальная логика**
   - Заглушка должна выполнять **только то, что необходимо для теста**.
   - Не стоит дублировать реальную бизнес-логику.
2. **Предсказуемость результатов**
   - Заглушка должна возвращать **фиксированные или параметризованные результаты** для разных входных данных.
3. **Изоляция тестируемого кода**
   - Заглушка должна полностью заменять внешнюю зависимость, чтобы тест был независимым.
4. **Простота и читаемость**
   - Код заглушки должен быть простым и понятным.
5. **Повторное использование**
   - Заглушки, часто используемые в тестах, лучше вынести в отдельные утилитарные классы или методы.
6. **Поддержка сценариев ошибок**
   - Можно создавать заглушки, которые выбрасывают исключения, чтобы тестировать обработку ошибок.
7. **Совместимость с инструментами тестирования**
   - Использовать Mockito, EasyMock или PowerMock для гибкого создания заглушек и моков.

---

1. **Что такое unit-тестирование?**
2. **Что такое заглушка (stub) в контексте тестирования ПО?**
3. **Какие преимущества дают фреймворки для модульного тестирования?**

---

1. Что такое unit-тестирование?

Unit-тестирование (модульное тестирование) — это проверка **отдельных единиц кода** (методов или классов) на корректность работы.

Основные цели:

1. Проверить правильность реализации функции или метода.
2. Изолировать тестируемый модуль от зависимостей (других классов, БД, сервисов).
3. Обеспечить раннее обнаружение ошибок на уровне кода.

Пример:

```
class Calculator {
    int sum(int a, int b) { return a + b; }
}

@Test
void testSum() {
    Calculator calc = new Calculator();
    assertEquals(5, calc.sum(2,3)); // проверяем корректность метода sum
}
```

---

2. Что такое заглушка (stub) в контексте тестирования ПО?

Stub (заглушка) — это **искусственный объект**, заменяющий реальную зависимость, и возвращающий заранее определённые данные.

Цель использования:

- Изолировать тестируемый компонент от внешних сервисов.
- Обеспечить предсказуемое поведение зависимостей.
- Ускорить выполнение тестов.

Пример с заглушкой:

```
UserRepository stubRepo = new UserRepository() {
    @Override
    public User findById(int id) {
        return new User(id, "Test User"); // фиксированный результат
    }
};
```

3. Преимущества фреймворков для модульного тестирования

Фреймворки, такие как **JUnit, TestNG, Mockito**, дают несколько ключевых преимуществ:

1. **Стандартизированная структура тестов**
   - Аннотации `@Test`, `@BeforeEach`, `@AfterEach` упрощают организацию тестов.
2. **Автоматизация тестирования**
   - Тесты можно запускать из IDE, build-tools (Maven/Gradle) и CI/CD.
3. **Assertions**
   - Предоставляют удобные методы для проверки результатов (`assertEquals`, `assertTrue`, `assertThrows`).
4. **Изоляция зависимостей**
   - С помощью заглушек (stubs) и моков (mocks) можно тестировать модули отдельно.
5. **Параметризованные тесты**
   - Позволяют запускать один тест с разными входными данными.
6. **Отчётность**
   - Генерация отчётов о пройденных и упавших тестах.

---

1. **В чем разница между заглушкой (stub) и макетом (mock)?**
2. **Как можно использовать макеты (mocks) для тестирования взаимодействия между компонентами?**
3. **Какие существуют библиотеки для создания заглушек и макетов в Java?**

---

1. Разница между заглушкой (stub) и макетом (mock)

Характеристика	Заглушка (Stub)	Макет (Mock)
Цель	Возврат заранее заданных данных для теста	Проверка взаимодействия с объектом (вызывов методов, параметров)
Логика	Минимальная или отсутствует	Может включать проверку порядка вызовов, количества вызовов
Применение	Тестирование **результата метода**	Тестирование **взаимодействия компонентов**
Пример	Метод `findUserById()` всегда возвращает `User("Test")`	Проверка, что метод `save()` вызван один раз с нужным объектом
Инструменты	Можно вручную или через Mockito	Mockito, EasyMock, JMock

Итог:

- **Stub** → изолирует тестируемый компонент и возвращает фиксированные данные.
- **Mock** → проверяет, как тестируемый компонент **взаимодействует с зависимостями**.

---

2. Использование макетов (mocks) для тестирования взаимодействия

Пример с Mockito:

```
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

class UserServiceTest {

    @Test
    void testUserSave() {
        UserRepository mockRepo = Mockito.mock(UserRepository.class);
        UserService service = new UserService(mockRepo);

        User user = new User("Alice");
        service.saveUser(user);

        // Проверяем, что метод save() был вызван ровно один раз с объектом
user
        Mockito.verify(mockRepo, Mockito.times(1)).save(user);
    }
}
```

Особенности:

- Проверяется **вызов метода**, а не сам результат.
- Позволяет тестировать **взаимодействие между слоями** (например, сервис и репозиторий).
- Можно имитировать ошибки или исключения и проверить обработку.

---

3. Библиотеки для создания заглушек и макетов в Java

Библиотека	Назначение	Особенности
Mockito	Создание stub и mock объектов	Простая интеграция с JUnit/TestNG, поддержка spy, verification
EasyMock	Mock объектов и проверка вызовов	Интерфейсно-ориентированная библиотека, немного более строгая
PowerMock	Расширение Mockito/EasyMock	Позволяет мокировать static методы, final классы, private методы
JMockit	Mock/stub и кодовые покрытия	Поддерживает тестирование сложных зависимостей, покрытие кода
Spring Boot Test	MockBean, для Spring компонентов	Интеграция с контекстом Spring, легко подменяет зависимости

1. **Какие принципы следует учитывать при проектировании тестов, использующих заглушки и макеты?**
2. **Какие подходы и методологии могут быть использованы для тестирования ПО в микросервисной архитектуре?**
3. **Как реализовать параметризованные тесты в JUnit для проверки поведения с различными входными данными?**

---

1. Принципы проектирования тестов с использованием заглушек (stubs) и макетов (mocks)

При использовании заглушек и макетов важно следовать нескольким принципам:

1. **Изоляция тестируемого кода**
   - Тест должен проверять только один компонент или метод, без влияния внешних зависимостей.
2. **Минимальная логика в заглушках**
   - Заглушки должны просто возвращать заранее заданные данные.
3. **Ясность и предсказуемость**
   - Результаты теста должны быть предсказуемыми, легко воспроизводимыми.
4. **Тестирование взаимодействий с моками**
   - Используйте **mock** для проверки вызовов методов, порядка вызовов и параметров.
5. **Использование параметризированных тестов**
   - Проверяйте различные варианты входных данных, чтобы увеличить покрытие кода.
6. **Поддержка сценариев ошибок**
   - Создавайте заглушки или моки, которые имитируют исключения, сбои API или недоступность сервисов.
7. **Повторное использование тестовых объектов**
   - Часто используемые заглушки/моки лучше вынести в отдельные классы или методы.

---

2. Подходы и методологии тестирования ПО в микросервисной архитектуре

В микросервисах тестирование требует комплексного подхода:

1. **Unit Testing (модульное тестирование)**
   - Проверяет отдельные классы и методы.
   - Используются **stubs и mocks** для изоляции.
2. **Integration Testing (интеграционное тестирование)**
   - Проверка взаимодействия между сервисами и внешними компонентами.
   - Инструменты: Spring Boot Test, TestContainers, Embedded DB.
3. **Contract Testing (тестирование контрактов)**
   - Проверка соответствия API между сервисами.
   - Инструменты: Pact, Spring Cloud Contract.
4. **End-to-End (E2E) Testing**
   - Полное тестирование бизнес-сценариев в системе.
   - Инструменты: Selenium, Cypress, Postman/Newman.
5. **Consumer-Driven Testing**
   - Тестирование сервиса на основе ожиданий потребителей (клиентов API).
6. **Load/Performance Testing**
   - Проверка работы под высокой нагрузкой.
   - Инструменты: JMeter, Gatling.

---

3. Реализация параметризованных тестов в JUnit

JUnit 5 позволяет запускать один тестовый метод с разными входными данными.

Пример с `@ParameterizedTest`

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;
import static org.junit.jupiter.api.Assertions.assertTrue;

class MathTest {

    @ParameterizedTest
    @ValueSource(ints = {2, 4, 6, 8})
    void testEvenNumbers(int number) {
        assertTrue(number % 2 == 0);
    }
}
```

Другие поставщики данных

Поставщик	Описание
`@ValueSource`	Массив простых значений (int, String, etc.)
`@CsvSource`	Несколько значений в формате CSV
`@CsvFileSource`	Данные из CSV файла
`@MethodSource`	Метод, возвращающий поток значений
`@EnumSource`	Перечисления (enum)

Особенности:

- Один тест выполняется несколько раз с разными входными данными.
- Позволяет проверять разные сценарии без дублирования кода.

---

1. **Что такое TDD и для чего оно используется?**
2. **Какие основные этапы процесса TDD?**
3. **В чем преимущества использования TDD при разработке ПО?**

---

1. Что такое TDD и для чего оно используется

TDD (Test-Driven Development) — это подход к разработке ПО, при котором **тесты пишутся до написания самого кода**.

Цель TDD:

1. Обеспечить правильность работы кода с самого начала.
2. Разрабатывать функционал **по шагам**, проверяя каждый сценарий.
3. Уменьшить количество багов и повысить качество кода.

Ключевая идея: сначала **тесты**, потом **код**, который заставляет тест пройти.

---

2. Основные этапы процесса TDD

Процесс TDD описывается циклом **Red → Green → Refactor**:

1. **Red (Написать тест, который падает)**
   - Пишется тест, проверяющий нужную функциональность.
   - Тест **не проходит**, так как код ещё не реализован.
2. **Green (Написать минимальный код, чтобы тест прошёл)**
   - Реализуется код только настолько, чтобы пройти тест.
   - Лишняя логика не добавляется.
3. **Refactor (Рефакторинг кода)**
   - Оптимизация и улучшение структуры кода без изменения функциональности.
   - Тесты продолжают проходить → гарантия, что функциональность сохранена.

Этот цикл повторяется для каждой новой функции или сценария.

---

3. Преимущества использования TDD

1. **Повышение качества кода**
    - Ошибки выявляются на раннем этапе.
2. **Меньше багов при добавлении нового функционала**
    - Тесты служат «защитой» при рефакторинге и изменениях.
3. **Лучшее проектирование кода**
    - Код становится модульным и легко тестируемым.
4. **Документация для разработчиков**
    - Тесты показывают, как должен работать код и какие сценарии поддерживаются.
5. **Упрощение интеграции**
    - Меньше неожиданностей при взаимодействии модулей.
6. **Повышение уверенности в изменениях**
    - Разработчик уверен, что функционал не сломается после изменений.

---

1. **Как TDD влияет на дизайн и архитектуру ПО?**
2. **Можете ли вы описать цикл "Красный-Зеленый-Рефактор" в контексте TDD?**
3. **Какие существуют распространенные ошибки при использовании TDD?**

---

1. Как TDD влияет на дизайн и архитектуру ПО

TDD оказывает значительное влияние на архитектуру и качество кода:

1. **Модульность и низкая связность**
   - Поскольку тесты пишутся для отдельных компонентов, код автоматически проектируется как независимые, изолированные модули.
2. **Высокая тестируемость**
   - Код становится проще для написания тестов, что часто требует явного выделения зависимостей (Dependency Injection).
3. **Чистая архитектура и интерфейсы**
   - Разработчик вынужден думать о **контрактах и интерфейсах**, а не о реализации.
4. **Раннее выявление проблем проектирования**
   - Если тестировать сложно → возможно, код слишком сложный или имеет плохую структуру.
5. **Упрощение рефакторинга**
   - С тестами легче изменять архитектуру без страха сломать функциональность.

2. Цикл "Красный → Зеленый → Рефактор" в TDD

Цикл состоит из трёх шагов:

1. Красный (Red)

- Пишется **тест для новой функции**.
- Тест **не проходит**, так как код ещё не реализован.
- Цель: определить требования и ожидаемый результат.

Пример:

```
@Test
void testSum() {
    Calculator calc = new Calculator();
    assertEquals(5, calc.sum(2, 3)); // тест падает, код ещё не написан
}
```

2. Зеленый (Green)

- Пишется **минимальный код**, чтобы тест прошёл.
- Не добавляется лишняя логика.
- Цель: сделать тест успешным.

```
class Calculator {
    int sum(int a, int b) {
        return a + b; // минимальная реализация
    }
}
```

3. Рефакторинг (Refactor)

- Улучшается структура кода без изменения функциональности.
- Все тесты должны продолжать проходить.

Пример:

```
// Вынесение общих методов, улучшение читаемости, соблюдение SOLID принципов
```

---

3. Распространённые ошибки при использовании TDD

Ошибка	Последствия	Как избежать
Слишком сложные тесты	Тесты трудно поддерживать, часто ломаются	Писать **простой и изолированный тест**
Игнорирование рефакторинга	Код становится грязным и трудно поддерживаемым	Всегда делать шаг Refactor после Green
Писать тесты после реализации	Потеря основной ценности TDD	Следовать циклу **Red → Green → Refactor**
Слишком много моков и заглушек	Тесты теряют связь с реальным кодом	Использовать моки только для внешних зависимостей
Не тестировать негативные сценарии	Скрытые ошибки остаются незамеченными	Добавлять тесты для исключений и ошибок
Игнорирование архитектуры	Тестируемый код плохо структурирован	Использовать TDD как инструмент для лучшего дизайна

---

1. **Как TDD влияет на поддержку и расширяемость ПО?**
2. **Как можно интегрировать TDD в существующий процесс разработки без значительных потерь в производительности?**
3. **Какие стратегии можно применить для обработки унаследованного кода, который не был разработан с использованием TDD?**

1. Влияние TDD на поддержку и расширяемость ПО

1. **Повышенная поддерживаемость**
   - Код хорошо протестирован, легко обнаруживать и исправлять ошибки.
   - Рефакторинг безопасен благодаря тестам, которые гарантируют сохранение функциональности.
2. **Лучшее разделение ответственности**
   - TDD стимулирует создание **модульного и изолированного кода**, что облегчает внесение изменений.
3. **Упрощённая интеграция новых функций**
   - Поскольку тесты охватывают существующий функционал, новые изменения можно вводить с минимальным риском сломать систему.
4. **Документация через тесты**
   - Тесты служат живой документацией: показывают, как использовать методы и чего ожидать от функций.
5. **Легкость адаптации к новым требованиям**
   - Архитектура становится более гибкой, так как каждый компонент изолирован и тестируем.

2. Интеграция TDD в существующий процесс разработки

Чтобы внедрить TDD без значительных потерь производительности:

1. **Постепенное внедрение**
   - Начинать с **новых функций** или новых модулей, не переписывая весь код сразу.
2. **Обучение команды**
   - Провести обучение TDD, показать практические примеры, шаблоны тестов.
3. **Использование быстрых тестов**
   - Модульные тесты должны выполняться **мгновенно**, чтобы не замедлять процесс сборки.
4. **CI/CD интеграция**
   - Настроить автоматический запуск тестов при каждом коммите или pull request.
5. **Выбор приоритетных участков**
   - Сначала тестировать критически важные части приложения, потом расширять покрытие.

3. Стратегии для работы с унаследованным кодом (legacy code)

Унаследованный код часто не имеет тестов и не был разработан с TDD. В таких случаях можно:

1. **Characterization Tests (тесты характеристик)**
   - Пишем тесты, которые **фиксируют текущее поведение** кода.
   - Позволяет безопасно рефакторить и добавлять тесты для новых функций.
2. **Incremental Refactoring**
   - Постепенно разбиваем большой код на модули с тестами.
   - Начинаем с критичных или часто изменяемых частей.
3. **Mocks и Stubs для изоляции**
   - Для сложных зависимостей используем заглушки и моки, чтобы тестировать отдельные части.
4. **Wrapper или Adapter**
   - Оборачиваем унаследованный код в слой, где можно писать TDD-тесты, не трогая оригинал.
5. **Покрытие тестами новых функций**
   - Любые новые функции пишем сразу с TDD, постепенно расширяя культуру тестирования на весь проект.