**Коллоквиум ПП Глазко Александр 8 группа**

**Вариант 3**

**1.3 Расписать что из себя представляет Decorator Design Pattern. Нарисовать «псевдо» UML диаграмму. Расписать какие аспекты инкапсулирует DP, какие аспекты изменчивости и сложности, привести примеры.**

Decorator Design Pattern - это паттерн, который позволяет динамически добавлять новые функции или изменять поведение объекта, не изменяя его класс. Это достигается путем создания нового класса-декоратора, который содержит ссылку на оригинальный объект и добавляет к нему новую функциональность.

UML диаграмма:



Decorator DP инкапсулирует аспекты изменчивости и сложности, так как позволяет добавлять новую функциональность без изменения исходного кода, что упрощает поддержку и расширение программы. Он также инкапсулирует аспекты композиции, так как использует композицию объектов для достижения цели.

Примеры использования Decorator DP:

- Добавление новых функций к базовому классу, не изменяя его кода.

- Добавление дополнительной функциональности к объектам во время выполнения программы.

- Использование в GUI-приложениях для динамического изменения внешнего вида элементов интерфейса.

**2.5 Что такое агрегация? Когда нужно применять наследование, а когда агрегацию?**

Агрегация - это форма отношений между объектами, когда один объект содержит ссылку на другой объект. Таким образом, один объект становится частью другого объекта.

Наследование и агрегация - это два разных подхода к организации кода.

Наследование используется, когда один класс наследует свойства и методы другого класса. Это означает, что наследующий класс расширяет функциональность базового класса, добавляя новые свойства и методы или переопределяя существующие.

Агрегация используется, когда один объект использует или содержит другой объект, но не является его расширением или не зависит от него. Он создает отношение "содержит" между объектами, где один объект является частью другого, но сохраняет свою самостоятельность.

Какой подход выбрать зависит от конкретной задачи и требований. В общем случае, наследование лучше подходит для организации отношений типа "является", когда объект наследует свойства и функциональность базового класса. Агрегация применяется в случаях, когда требуется связать два объекта, чтобы один мог использовать функциональность другого, но при этом сохранить их независимость.

**2.7 Что такое динамический и что такое статический полиморфизм?**

Полиморфизм в программировании - это возможность объектов разных классов использовать одну и ту же функцию с одинаковым именем, но с разными реализациями.

Статический полиморфизм, или перегрузка функций, происходит во время компиляции. Компилятор определяет, какая функция будет вызываться на основе типов аргументов и параметров. Например, у вас может быть функция с именем "print", которая может принимать аргументы разных типов, таких как int, float или string, и в зависимости от типа аргумента, компилятор будет выбирать правильную реализацию функции.

Динамический полиморфизм, или наследование и виртуальные функции, происходит во время выполнения программы. Когда у вас есть базовый класс и производные классы, содержащие одноименные виртуальные функции, объекты этих классов могут вызывать одну и ту же функцию с одинаковым именем, но с разными реализациями. Компилятор отслеживает тип объекта во время выполнения и вызывает правильную реализацию виртуальной функции.

Например, у вас может быть базовый класс "Shape" и производные классы "Circle" и "Rectangle". У этих классов может быть одинаковая виртуальная функция с именем "area", которая возвращает площадь фигуры. Когда вы создаете объекты этих классов и вызываете функцию "area", компилятор вызывает правильную реализацию функции в зависимости от типа объекта, то есть если вы вызываете "area" для объекта типа "Circle", будет вызвана реализация функции "area" для класса "Circle".

Оба вида полиморфизма - статический и динамический - помогают сделать код более гибким и модульным, но применяются в разных ситуациях.

**2.10 Почему ООП изобрел Сократ (в изложении Платона), ввели в IT в 60-е годы, а начали применять в 90-е?**

Основная концепция объектно-ориентированного программирования (ООП) была представлена еще в древнегреческой философии Сократом (через изложение Платона), который развивал идеи о существовании абстрактных понятий и классов. Однако, конкретное применение ООП в программировании началось значительно позднее.

Официальное введение ООП в IT-сфере произошло в 60-х годах XX века. В 1967 году американские ученые особое внимание уделили теме объектно-ориентированного программирования и появилась первая попытка создать язык программирования, основанный на принципах ООП - симула (Simula). Однако, в то время компьютеры были медленными и имели ограниченные ресурсы, что не позволяло широко использовать и эффективно работать с ООП.

Несмотря на это, в 90-х годах произошел резкий рост мощности компьютеров и развитие программного обеспечения, что создало благоприятные условия для применения ООП. Одним из наиболее популярных языков программирования, основанных на принципах ООП, стал Java, который был представлен в 1995 году. Появление Java и других современных языков программирования, поддерживающих ООП-парадигму, стимулировало широкое применение ООП и его активное развитие в 90-х годах и в последующие годы.

**2.2 Что такое ООП декомпозиция? Приведите примеры.**

ООП декомпозиция (или объектно-ориентированная декомпозиция) - это методологический подход к разбиению программного проекта на более мелкие, понятные и управляемые элементы. При использовании ООП декомпозиции большие задачи разбиваются на более мелкие, взаимодействующие между собой объекты, которые легче понимать и поддерживать.

Примеры ООП декомпозиции:

1. Разработка системы для управления банковским счетом. Вместо создания одного большого класса, который реализует все функции банковского счета, можно разбить его на несколько классов, таких как "Клиент", "Счет", "Транзакция" и "Банк". Каждый из этих классов будет отвечать за свою часть функциональности.

2. Разработка компьютерной игры. Вместо создания одного огромного класса "Игра", можно разбить ее на более мелкие классы, такие как "Игровой персонаж", "Игровая среда", "Уровень" и "Задачи". Каждый из этих классов будет отвечать за свою часть игровой логики.

3. Разработка онлайн-магазина. Вместо создания одного класса, реализующего все функции магазина, можно разбить его на классы, такие как "Товар", "Корзина", "Заказ" и "Пользователь". Каждый из этих классов будет отвечать за свою часть функциональности магазина.

ООП декомпозиция помогает улучшить структуру и понятность программного проекта, а также облегчает его поддержку и расширение в будущем.

**3.6 Что такое Test Runner, дайте определение, приведите примеры Java Frameworks.**

Test Runner - это инструмент, используемый для запуска и управления автоматизированными тестами. Он выполняет тесты в определенном порядке и предоставляет отчеты о результатах выполнения тестов.

Примеры Test Runner в Java:

1. JUnit - один из самых популярных Java фреймворков для написания модульных тестов. В JUnit используется собственный Test Runner для запуска тестов и проверки их результатов.

2. TestNG - еще один популярный Java фреймворк для автоматизированного тестирования. TestNG имеет свой собственный Test Runner, который позволяет гибко настраивать запуск и управление тестами.

3. Cucumber - фреймворк, используемый для функционального тестирования на основе Behavior-Driven Development (BDD). Cucumber также имеет свой Test Runner для запуска тестов, описанных в сценариях на языке Gherkin.

Определение Java frameworks - это набор инструментов и классов, предназначенных для упрощения и ускорения разработки на языке Java. Java frameworks предоставляют готовые решения для решения типичных задач, таких как управление базами данных, разработка веб-приложений, тестирование и другие. Эти фреймворки предлагают специальные API, синтаксис и стандарты, которые облегчают разработку и повышают эффективность проекта. Некоторые из известных Java фреймворков включают Spring, Hibernate, Apache Struts, JavaFX и другие.

**3.8 Что такое Mock Framework? Приведите хотя бы 4 случая при которых необходимо использовать Mock-и.**

Mock Framework (или фреймворк для мокирования) - это инструмент или библиотека, которая позволяет создавать и управлять мок-объектами (или заглушками) в тестовом окружении. Моки представляют собой замену реальным объектам и используются для изоляции тестируемого кода от зависимых компонентов. Вот четыре случая, когда полезно использовать моки:

1. Изоляция зависимостей: Когда вы хотите изолировать тестируемый объект от внешних зависимостей, таких как базы данных, внешние службы или сетевые вызовы. Моки позволяют имитировать поведение этих зависимостей.

2. Тестирование ошибочных сценариев: Вы можете использовать моки, чтобы создать ситуации, которые в реальности могут быть сложно воспроизвести, например, ошибки базы данных или сетевые сбои, и проверить, как ваше приложение реагирует на такие события.

3. Улучшение производительности тестов: Заменяя реальные службы или компоненты моками, вы можете ускорить выполнение ваших тестов, так как моки могут имитировать вызовы быстрее, чем реальные компоненты.

4. Создание специфических сценариев: Моки позволяют вам создавать специфические сценарии и условия для тестирования, которые могут быть сложными или дорогостоящими в реальных условиях, такие как тестирование обработки ошибок, асинхронных событий и т. д.

Использование моков в тестировании помогает создавать более надежные и предсказуемые тесты, упрощая проверку ваших приложений в разных сценариях.

**3.11 Что такое Annotation в Java? Как используется в контексте Mock frameworks?**

Аннотация (Annotation) в Java представляет собой специальный тип метаданных, которые можно добавлять к классам, методам, полям и другим элементам кода. Аннотации предоставляют дополнительную информацию о коде и могут быть использованы компилятором или другими инструментами для выполнения определенных действий.

В контексте Mock frameworks, аннотации используются для определения поведения заглушек (mock objects) и настройки тестового окружения. Например, в Mockito можно использовать аннотацию @Mock для создания заглушек объектов:

java

@Mock

List<String> mockList;

Эта аннотация указывает Mockito на необходимость создания заглушки для интерфейса List с типом элементов String. Затем можно использовать эту заглушку в тестовом коде без явного создания объекта:

java

@Test

public void testMethod() {

// Использование заглушки

mockList.add("example");

// Проверка вызова метода на заглушке

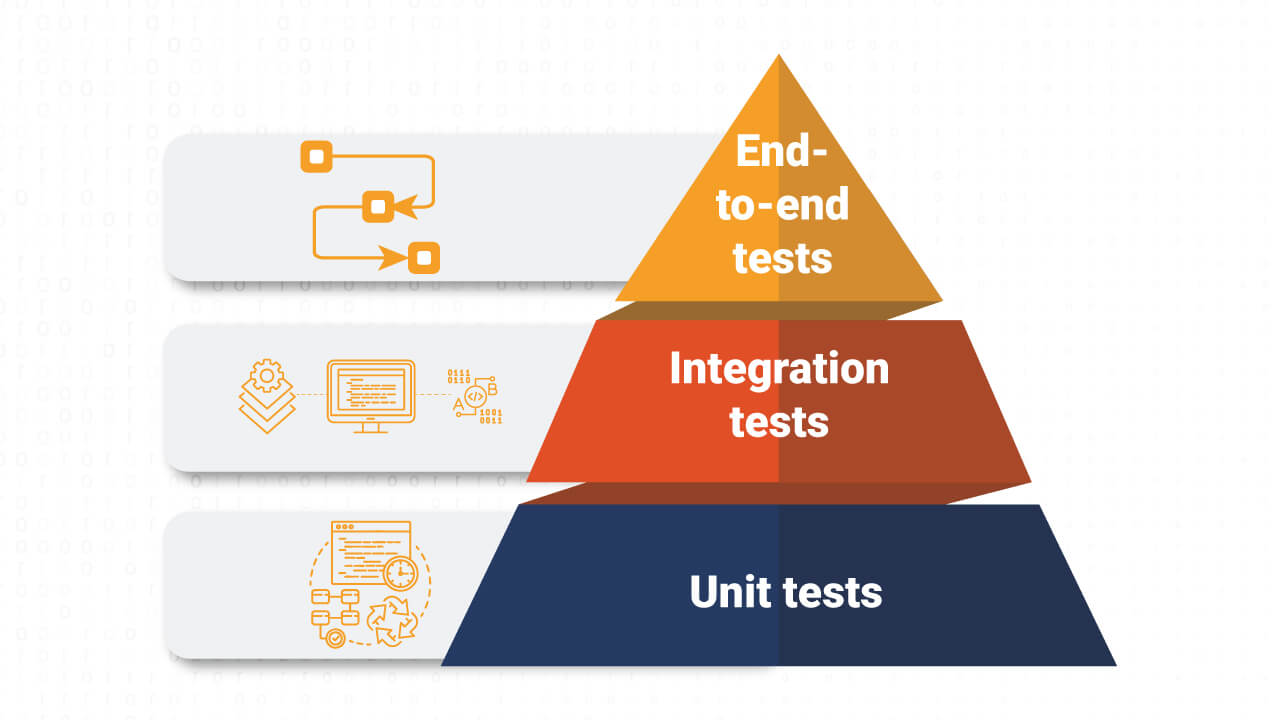
Mockito.verify(mockList).add("example");

}

Аннотация @Mock позволяет более удобно создавать заглушки и автоматически интегрируется с Mock frameworks, чтобы обеспечить правильное поведение и настройку заглушек в контексте тестирования.

**3.1 Пирамида тестирования? Определение, физический смысл, примеры.**

Пирамида тестирования - это стратегия тестирования, которая предлагает разбить тесты на группы разной детализации и количества. Она помогает определить, какие тесты следует выполнять чаще, а какие - реже. По мере увеличения детализации количество тестов должно уменьшаться. Вот пример пирамиды тестирования:

Примеры:

- Unit-тесты

- Интеграционные тесты

- UI-тесты

**4.1 Реализовать в ООП стиле следующую задачу: рассчитать первых n факториалов, вернуть контейнер с первыми n факториалами. Код должен быть промышленного качества: обрабатывать ошибки, исключительные ситуации, покрыт UnitTest-ами.**

Лежит на GitHub, в целом как и весь коллоквиум 😊

**5.1 Реализовать Абстрактную Фабрику, можно фактически «скопировать» «реализацию с доски» (кто не ходил, тем хуже). Покрыть код UnitTest-ами.**

Лежит на GitHub, в целом как и весь коллоквиум 😊