КОЛЛОКВИУМ БПИ228

КПО 2023-2024

2023 декабрь

1 Основы языка UML

1.1. Назначение UML. (Что такое UML и каково его основное назначение?)

UML (*Unified Modeling Language*) - это формальный язык для визуализации, спецификации, конструирования и документирования системного дизайна программного обеспечения. Основное назначение UML заключается в предоставлении стандартных нотаций и инструментов для моделирования и описания различных аспектов системы.

UML имеет широкое применение в области разработки программного обеспечения, позволяя командам разработчиков и заинтересованным сторонам лучше понимать требования проекта, проектировать архитектуру системы, моделировать бизнес-процессы и обмениваться информацией между различными участниками проекта.

1.2. Основные виды диаграмм UML (Какие основные типы диаграмм используются в UML? Объясните, для чего эти диаграммы применяются.)

Типы диаграмм UML:

- Диаграммы классов:
  - Предоставляют визуальное представление классов, атрибутов, методов и отношений между классами. Используются для проектирования и анализа структуры системы.

- Диаграммы прецедентов:
  - Описывают взаимодействие между акторами и функциональностью системы. Используются для анализа требований и определения функциональности системы.
- Диаграммы последовательностей:
  - Показывают взаимодействие между объектами в определенной последовательности шагов или временной линии. Используются для моделирования поведения системы и детализации последовательности выполнения операций.
- Диаграммы состояний:
  - Описывают различные состояния, в которых может находиться объект или система, и переходы между этими состояниями. Полезны для моделирования поведения системы и состояний объектов в процессе выполнения.
- Диаграммы компонентов:
  - Показывают различные компоненты системы и их зависимости. Используются для проектирования архитектуры системы и модуляризации компонентов.
- Диаграммы развертывания:
  - Описывают физическое размещение компонентов системы и их взаимодействие в распределенных окружениях. Полезны при планировании и конфигурировании системы.

1.3. Диаграммы UML (Какой функционал необходимо описать при моделировании системы заказов доставки посылок?)

При моделировании системы заказов доставки посылок, вы можете рассмотреть следующие основные функциональные элементы:

1. Создание заказа:
   - Возможность создания нового заказа с указанием информации о отправителе, получателе, весе и размерах посылки.

- Валидация и проверка введенных данных для достоверности информации.

2. Расчет стоимости и выбор способа доставки:

   - Расчет стоимости доставки на основе факторов, таких как расстояние, вес, тип доставки (экспресс, стандарт и т. д.), дополнительные услуги и тарифы.
   - Предоставление вариантов способов доставки (наземный, авиа, морской и т. д.), каждый из которых имеет разные ограничения по времени и стоимости.

3. Отслеживание статуса заказа:

   - Возможность отслеживания статуса заказа на каждом этапе доставки, начиная с приема заказа, подготовки к отправке, транзитных точек, доставки и получения посылки.
   - Уведомления или оповещения для отправителя и получателя о текущем статусе заказа (по электронной почте, SMS и т. д.).

4. Управление сроками доставки:

   - Предоставление информации о предполагаемом времени доставки в зависимости от выбранного способа и типа доставки.
   - Учет возможных задержек (погодные условия, пробки и другие внешние факторы) и предоставление обновленного времени доставки.

5. Управление оплатой и финансовыми операциями:

   - Предоставление различных методов оплаты (кредитные карты, электронные платежи и др.).
   - Обработка платежей, включая расчет стоимости доставки, учет налогов и сборов, выставление счетов и генерацию квитанций.

6. Управление информацией о клиентах и адресной книгой:

   - Хранение и управление базой данных клиентов с информацией о контактах, адресах и предпочтениях доставки.
   - Возможность сохранения адресов отправителя и получателя для повторного использования при создании новых заказов.

7. Генерация отчетов и аналитика:

- Составление отчетов о выполненных доставках, выручке, популярности определенных услуг и анализ других данных для принятия управленческих решений или оптимизации процессов.

1.4. Диаграммы классов UML
(Кратко опишите, какие классы необходимы для моделирования системы заказов посылок? Какие связи необходимы между классами?)

Классы и связи

Класс "Пользователь"(User)

Этот класс представляет пользователей системы, которые размещают заказы на посылки. Он может содержать атрибуты, такие как идентификатор пользователя, имя, контактные данные и историю заказов.

Класс "Заказ"(Order)

Этот класс представляет отдельный заказ на посылку. Он может содержать атрибуты, такие как идентификатор заказа, дата размещения, адрес отправления и адрес доставки, а также статус заказа.

Класс "Посылка"(Parcel)

Этот класс представляет собой конкретную посылку, которая относится к определенному заказу. Он может содержать атрибуты, такие как идентификатор посылки, тип товара, размер, вес и стоимость доставки.

Класс "Курьер"(Courier)

Этот класс представляет курьера или доставку, который транспортирует посылки от отправителя к получателю. Он может содержать атрибуты, такие как идентификатор курьера, имя, контактные данные и статус доступности.

Связи

Ассоциация между классами "Пользователь"и "Заказ"

Каждый пользователь может размещать несколько заказов, поэтому между этими классами может существовать связь "один ко многим".

Ассоциация между классами "Заказ"и "Посылка"

Каждый заказ может содержать несколько посылок, а каждая посылка относится к только одному заказу. Поэтому между этими классами может существовать связь "один ко многим"или "один к одному".

Ассоциация между классами "Заказ"и "Курьер"

Каждый заказ может быть доставлен различными курьерами, а курьер может иметь несколько заказов. Поэтому между этими классами может существовать связь "многие ко многим".

1.5. Диаграммы последовательностей UML (Кратко опишите диаграмму последовательностей для прецедента авторизации пользователя.)

- Актер "Пользователь"(*User*) отправляет запрос на авторизацию, вводя свои учетные данные в интерфейс системы.
- Интерфейс системы (*System Interface*) получает эти учетные данные и создает сообщение с запросом авторизации.
- Авторизационный контроллер (*Authentication Controller*) получает сообщение и выполняет процесс проверки учетных данных пользователя.
- Авторизационный контроллер связывается с базой данных (*Database*) для проверки соответствия учетных данных.
- База данных выполняет поиск и сравнивает предоставленные учетные данные с данными о пользователях, хранящимися в базе.
- База данных отправляет результат проверки обратно авторизационному контроллеру.

- Авторизационный контроллер создает сообщение с результатом авторизации и отправляет его обратно в интерфейс системы.
- Интерфейс системы получает результат авторизации и выполняет соответствующие действия, например, перенаправляет пользователя на главную страницу или отображает сообщение об ошибке.

1.6. Диаграммы активностей UML (Кратко опишите диаграмму активностей для прецедента авторизации пользователя.)

- Начало диаграммы обозначается стартовой точкой.
- Система отображает интерфейс авторизации и ожидает ввода учетных данных от пользователя.
- Пользователь вводит свои учетные данные в интерфейс.
- Система проверяет введенные учетные данные на валидность. Если данные неверные, переход на шаг 6.
- Система устанавливает сеанс авторизованной сессии для пользователя. Процесс авторизации завершается успешно.
- Система отображает сообщение об ошибке авторизации.
- Конец диаграммы обозначается конечной точкой.

2 Основы языка программирования Kotlin

2.1. Классы в Kotlin
(Зачем нужны классы? Как создаются классы в Kotlin? Что такое блок init? Что такое primary constructor? В каком порядке они вызываются?)

- Классы нужны для описания состояния и поведения объектов. Они позволяют создавать шаблоны для объектов.
- В Kotlin классы создаются с помощью ключевого слова `class`.
- Primary constructor (основной конструктор) объявляется в заголовке класса и может содержать параметры.
- Блок `init` в Kotlin представляет собой блок инициализации, выполняющийся при создании каждого экземпляра класса перед вызовом конструктора.
- Порядок вызова в Kotlin следующий:
    1. Блок инициализации (`init`) вызывается непосредственно перед вызовом основного конструктора.
    2. Основной конструктор вызывается после блока инициализации.

2.2. Свойства и поля в Kotlin
(Какими способами можно создать свойство в Kotlin? Зачем классам нужны свойства?)

- Свойства в `Kotlin` могут быть объявлены с помощью ключевых слов `val` (неизменяемое свойство) или `var` (изменяемое свойство).
- Классам нужны свойства для хранения и предоставления доступа к данным.
- Свойства могут иметь `геттеры` и `сеттеры`, которые позволяют контролировать доступ к данным.

2.3. Наследование в Kotlin
(Зачем применяют наследование? Как наследуются интерфейсы и классы в Kotlin?)

- Наследование применяется для создания иерархии классов, где классы-наследники наследуют свойства и методы от родительских классов.
- В `Kotlin` классы наследуются с помощью ключевого слова ":".
- Интерфейсы в `Kotlin` также могут наследоваться с помощью ключевого слова ":".

2.4: Основные структуры данных в Kotlin
(Какие основные структуры данных имеются в Kotlin. Зачем они нужны?)

В Kotlin доступны следующие основные структуры данных:

- Массивы (Arrays): Массивы представляют собой упорядоченные коллекции элементов определенного типа. Они позволяют эффективно хранить и обрабатывать наборы данных. Массивы полезны, когда требуется работать с фиксированным количеством элементов.
- Списки (Lists): Списки представляют собой коллекции элементов переменной длины, где элементы упорядочены и могут дублироваться. Списки предоставляют методы для добавления, удаления, изменения и доступа к элементам.
- Множества (Sets): Множества представляют собой коллекции элементов, в которых каждый элемент может встречаться только один раз. Они обеспечивают быструю проверку принадлежности элемента к множеству и операции объединения, пересечения и разности множеств.
- Словари (Maps): Словари представляют собой коллекции, где каждый элемент состоит из пары ключ-значение. Они позволяют быстро находить значение по ключу и обеспечивают операции добавления, удаления и изменения элементов.

`Эти структуры данных нужны для эффективной организации, хранения и обработки данных в приложении. Они предоставляют удобные методы`

для работы с коллекциями, позволяют упорядочивать, фильтровать и манипулировать данными, а также обеспечивают эффективный доступ и поиск элементов. Они являются важной частью разработки программ и помогают в создании эффективных и структурированных решений.

2.5. Null-безопасность в Kotlin (Что такое null-безопасность в Kotlin? Зачем она нужна?)

- Null-безопасность в Kotlin предназначена для предотвращения ошибок, связанных с нулевыми значениями.
- В Kotlin типы данных могут быть объявлены с использованием оператора ?, что позволяет им принимать значение null или быть ненулевыми.

2.6. Code style в Kotlin (Виды написания составных слов. В каких случаях используется тот или иной тип?)

Относительно видов написания составных слов в Kotlin, есть два основных типа:

- PascalCase (также известный как UpperCamelCase): Составные слова пишутся с заглавной первой буквой каждого слова, без использования пробелов или подчеркиваний. Например: MainActivity, HttpRequest, UserModel. Этот стиль обычно используется для именования классов и типов данных.
- CamelCase (также известный как lowerCamelCase): Составные слова пишутся с заглавной первой буквой первого слова, а остальные слова начинаются с заглавной буквы. Например: calculateTotal, getUserById, firstName. Этот стиль обычно используется для именования переменных, методов и свойств.
- SnakeCase: Составные слова разделяются знаком подчеркивания. Несмотря на то, что этот стиль менее распространен в Kotlin, он может использоваться в некоторых случаях, особенно если ваш проект имеет существующий код со стилем SnakeCase.

2.7. Лямбда-выражения в Kotlin (Что такое лямбда-выражение? Зачем они нужны? Что такое функция высшего порядка? Где в Kotlin они используются?)

- Лямбда-выражение - это компактный способ представления анонимной функции, которая может быть передана как аргумент или присвоена переменной. Они используются для создания функций "на лету"без явного объявления функции и имеют доступ к переменным в окружающей области видимости. Лямбда-выражения полезны, когда требуется передать функцию как параметр или когда требуется определить простую функциональность внутри другой функции.

- Функция высшего порядка - это функция, которая может принимать одну или несколько функций в качестве аргументов или возвращать функцию как результат. В Kotlin функции являются объектами первого класса, поэтому их можно передавать, присваивать переменным и возвращать из других функций. Функции высшего порядка позволяют писать более гибкий и модульный код, позволяют абстрагировать общий функционал и легко повторно использовать код.

- В Kotlin лямбда-выражения и функции высшего порядка активно используются во многих местах, включая:

    1. Функции высшего порядка могут быть использованы при работе с коллекциями данных, например, для фильтрации, сортировки или преобразования элементов коллекции.
    2. Они также используются в функциях, которые требуют колбэков или обратных вызовов, например, в асинхронных операциях или обработке событий.
    3. Лямбда-выражения позволяют определить функциональность непосредственно при вызове, что удобно при работе с потоками данных `(Stream API)` или операциями над коллекциями.
    4. `Kotlin` предоставляет удобный синтаксис для работы с функциями высшего порядка, в том числе встроенные функции, такие как `map, filter, reduce` и др. которые упрощают работу с коллекциями данных.

3 Принципы и концепции программирования

3.1. Принципы SOLID (Что представляют собой принципы SOLID в контексте программирования?)

Принципы SOLID (SOLID - аббревиатура от принципов Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation и Dependency Inversion) в контексте программирования являются набором принципов, которые помогают создавать гибкие, расширяемые и поддерживаемые программные системы.

- Принцип единственной ответственности (Single Responsibility Principle) - класс должен иметь только одну причину для изменения. Он должен быть ответственен только за один аспект функциональности.
- Принцип открытости/закрытости (Open-Closed Principle) - программные сущности (классы, модули, функции и т.д.) должны быть открыты для расширения, но закрыты для модификации. Это означает, что новый функционал должен добавляться путем расширения, а не изменения кода существующих компонентов.
- Принцип подстановки Лисков (Liskov Substitution Principle) - объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности программы. Это означает, что подклассы должны быть совместимы с базовым классом и сохранять его контракты.
- Принцип разделения интерфейса (Interface Segregation Principle) - клиенты не должны зависеть от интерфейсов, которые они не используют. Интерфейсы должны быть маленькими, специфичными и сфокусированными на конкретных клиентах.
- Принцип инверсии зависимостей (Dependency Inversion Principle) - модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба должны зависеть от абстракций. Абстракции не должны зависеть от деталей, а детали должны зависеть от абстракций. Это принцип подразумевает использование интерфейсов или абстрактных классов для определения зависимостей.

3.2. Принципы KISS и DRY (Объясните принципы KISS и DRY. Для чего сформулированы эти принципы?)

- Принцип KISS (Keep It Simple, Stupid) означает, что программный код должен быть простым и понятным. Вместо сложных и запутанных решений, код должен быть написан так, чтобы его легко понять и поддерживать.
- Принцип DRY (Don't Repeat Yourself) предписывает избегать дублирования кода. Если у нас есть повторяющиеся фрагменты кода, то их следует вынести в отдельные функции, классы или модули и использовать их повторно. Каждая часть кода должна иметь единственное источник истины.

3.3. Основные парадигмы программирования. (Каковы основные парадигмы программирования и в чем их отличительные особенности?)

Основные парадигмы программирования:

- Процедурное программирование: в этой парадигме программа разбивается на набор процедур, которые выполняют определенные действия. Основные концепции - последовательность, выбор и повторение.
- Объектно-ориентированное программирование (ООП): в этой парадигме программа организуется вокруг объектов, которые объединяют данные и функциональность в единую сущность. Основные концепции - инкапсуляция, наследование, полиморфизм и абстракция.
- Функциональное программирование: в этой парадигме основной упор делается на функции как основной строительный блок программы. Основные концепции - неизменяемость данных, отсутствие побочных эффектов и рекурсия.
- Логическое программирование: эта парадигма основана на формальной логике. Программа состоит из набора правил и фактов,

и выполнение программы сводится к выводу результата на основе этих правил и фактов.

3.4. Основы ООП. (Что такое ООП? Устно приведите пример реализации каждого принципа на Koltin. Каковы основные достоинства и недостатки этой парадигмы программирования?)

ООП (Объектно-ориентированное программирование) - это парадигма программирования, которая основана на концепции объектов. Она включает в себя следующие принципы:

- Инкапсуляция: объединение данных и функциональности внутри объекта и доступ к ним через интерфейс объекта.

```
class Person {
    private var name: String = ""

    fun setName(name: String) {
        this.name = name
    }

    fun getName(): String {
        return name
    }
}
```

- Наследование: процесс создания новых классов на основе существующих.

```
open class Animal {
    open fun makeSound() {
        println("The animal makes a sound.")
    }
}

class Dog : Animal() {
    override fun makeSound() {
        println("The dog barks.")
```

```
    }
}
```

- Полиморфизм: возможность объектов с одинаковым интерфейсом иметь различную реализацию.

```
interface Shape {
    fun draw()
}

class Circle : Shape {
    override fun draw() {
        println("Drawing a circle.")
    }
}

class Rectangle : Shape {
    override fun draw() {
        println("Drawing a rectangle.")
    }
}
```

- Абстракция: создание абстрактного представления объекта, которое определяет его важные характеристики и функциональность, но скрывает сложные детали.

```
abstract class Vehicle {
    abstract fun start()
}

class Car : Vehicle() {
    override fun start() {
        println("Starting the car.")
    }
}
```

Основные достоинства ООП включают модульность, повторное использование кода, упрощение сопровождения, улучшенную понятность и расширяемость.

Недостатки включают большую сложность в проектировании и обучении, возможные проблемы с производительностью и потребление большего объема памяти.