ВВЕДЕНИЕ В ОБЪЕКТНО ОРИЕНТИРОВАННЫЙ ДИЗАЙН С JAVA

Принципы и шаблоны объектно-ориентированного проектирования

18+

ТИМУР МАШНИН

Тимур Машнин

Введение в объектно-ориентированный дизайн с Java

http://www.litres.ru/pages/biblio_book/?art=67289136
SelfPub; 2022

Аннотация

Эта книга ориентирована на тех, кто уже знаком с языком программирования Java и хотел бы углубить свои знания и изучить объектно-ориентированный анализ и проектирование программного обеспечения. Вы познакомитесь с основными принципами и паттернами объектно-ориентированного дизайна, используемыми при разработке программных систем Java. Вы научитесь моделировать системы Java с помощью UML диаграмм, познакомитесь с основными понятиями и принципами объектно-ориентированного подхода, изучите порождающие, структурные и поведенческие шаблоны проектирования. Вы узнаете, как создавать модульное, гибкое и многоразовое программное обеспечение, применяя объектно-ориентированные принципы и шаблоны проектирования.

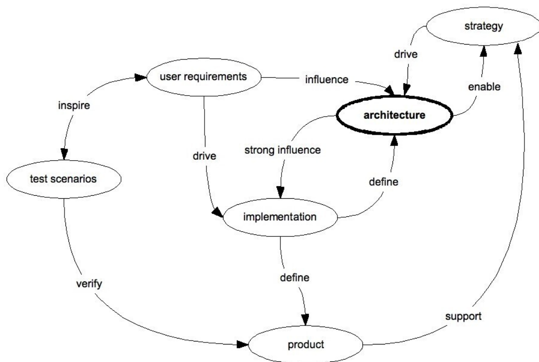
Содержание

Введение 4
Вопросы 24
Основные понятия 29
Принципы ООД (Объектно-ориентированного дизайна) 55
Принцип Абстракции в UML 79
Принцип Инкапсуляции в UML 86
Конец ознакомительного фрагмента. 89

Тимур Машнин
Введение в объектно-ориентированный дизайн с Java

Введение

Что такое дизайн и архитектура программного обеспечения?



И как это может улучшить программные продукты?

Давайте рассмотрим сценарий.

Предположим, вы присоединяетесь к проекту, который находится уже в разработке некоторое время.

Вы смотрите на код проекта, и вы не можете понять, для чего предназначены эти куски кода, так как он плохо организован, и проектной документации не существует.

Вы даже не знаете, с чего начать.

Это все признаки того, что проект не был хорошо разработан с самого начала.

Или, допустим, вы сейчас работаете над персональным проектом.

Когда вы начинали, вы не были уверены, какая конкретно функциональность должна быть реализована, но тем не менее вы начали кодирование.

Для вас не имело значения, что код будет неорганизованным, потому что вы были единственным, кто работал над проектом.

И предположим, вы придумали замечательную новую функцию для своего продукта, но при ее реализации вы нарушили программу в других местах. И теперь вы должны все исправлять во многих местах своего кода.

Чего не произошло бы, если бы вы правильно и хорошо с самого начала спроектировали бы свой продукт.

И такие сценарии довольно часто встречаются в индустрии программного обеспечения, что показывает, почему

дизайн и архитектура программного обеспечения так полезны.

В этом разделе вы узнаете, как применять принципы и паттерны дизайна и архитектуры для создания многоразовых и гибких программных систем. Вы узнаете, как задокументировать дизайн и архитектуру программного продукта визуально.

Итак, в чем разница между дизайном программного обеспечения и архитектурой программного обеспечения?

Роль дизайнера программного обеспечения или архитектора программного обеспечения может сильно отличаться от компании к компании.

На это влияют такие характеристики, как размер компании, объем проекта, опыт команды разработчиков, организационная структура и возраст компании.

В некоторых компаниях могут работать отдельные дизайнеры или архитекторы.

В других компаниях эта работа может выполняться членом или членами команды разработчиков.

И как правило, дизайнер программного обеспечения отвечает за определение программного решения для конкретной проблемы путем проектирования деталей отдельных компонентов и их обязанностей.

Дизайнер программного обеспечения отвечает за просмотр всей системы и выбор подходящих фреймворков, систем хранения данных, за решения и определения взаимо-

действий компонентов друг с другом.

И это подводит нас к основному различию между дизайном программного обеспечения и архитектором программного обеспечения.

Дизайнер программного обеспечения смотрит на аспекты системы более низкого уровня, тогда как архитектор программного обеспечения работает с более крупной картиной – с более высокими аспектами системы.

Подумайте об этом, как о проектировании здания.

Архитектор сосредотачивается на основных структурах и службах, в то время как дизайнер интерьера фокусируется на меньших пространствах внутри здания.

Дизайн программного обеспечения – это процесс превращения пожеланий и требований заказчика в рабочий код, который является стабильным и поддерживаемым в долгосрочной перспективе, и может быть развит и стать частью более крупной системы.

Архитектура программного обеспечения в первую очередь начинается с понимания того, в чем состоит бизнес-задача, которую должен решить клиент.

И основная задача заключается в том, чтобы выяснить, чего хочет клиент, тогда можно двигаться дальше.

Потому что, если вы понимаете задачу, вы можете начать думать о возможных решениях, а затем вы начинаете понимать, как будет выглядеть общее решение.

И архитектура важна, потому что, если вы ошибетесь, ваш

проект не удастся.

Все просто.

Мы знаем это в области строительства, и мы это знаем в области программного обеспечения.

Архитектура – это понимание взаимосвязи между требованиями пользователя и способностью создавать систему, которая будет обеспечивать эти требования.

При этом самая большая проблема, с которой мы сталкиваемся, – это понимание проблемы клиента.

Что он действительно хочет сделать?

И во многих случаях клиент фактически не знает, что он хочет делать. Он приходит лишь с частичным пониманием, смутным чувством, что он может сделать что-то лучше.

И одна из первых задач состоит в том, чтобы помочь ему лучше понять его проблему.

Задача архитекторов программного обеспечения – это взаимодействие между продуктом, клиентом и инженерными командами.

Архитектор программного обеспечения похож на архитектора здания. И он отвечает за общую концептуальную целостность проекта.

Возможно, вы слышали термин «объектно-ориентированное моделирование».

Что это?

При решении задачи, объектно-ориентированное моделирование включает в себя практику представления ключевых

понятий через объекты в вашем программном обеспечении.

И в зависимости от задачи многие концепции становятся отдельными объектами в программном обеспечении.

Подумайте об объектах.

Вокруг нас все объекты.

Почему вы должны использовать объекты для представления вещей в вашем коде?

Это способ держать ваш код организованным, гибким и многоразовым.

Объектный подход создает организованный код, содержа связанные детали и конкретные функции в разных, легко доступных местах.

Это создает гибкость, поскольку вы можете легко изменять детали модульным способом, не затрагивая остальную часть кода. Также вы можете повторно использовать разные части кода.

Давайте рассмотрим, как может выглядеть объектно-ориентированное моделирование.

Рассмотрим, например, помещение для семинаров.

Первый объект, который мы идентифицируем, является сама комната.

В комнате есть такие детали, как номер комнаты и места для сидения.

Также мы можем идентифицировать объекты, которые содержатся в этой комнате.

Существует множество физических объектов, такие как

стул, стол, проектор и белая доска.

Каждый из этих физических объектов может быть представлен объектами в программном обеспечении.

И существуют конкретные детали, связанные с каждым объектом.

Проектор имеет характеристики, связанные с его производительностью, такие как разрешение и яркость.

И объекты также могут иметь индивидуальные обязанности или поведение.

Например, проектор принимает видеопоток и отображает изображение.

Вы можете думать о разработке программного обеспечения как о процессе, который берет задачу и создает решение с помощью программного обеспечения.

И как правило, это итеративный процесс, при этом каждая итерация берет набор требований для реализации и тестирования и в конечном итоге создается полное решение.

Многие разработчики стремятся сразу кодировать, несмотря на то, что не полностью понимают, что программировать в первую очередь.

И погружение прямо в работу по реализации является основной причиной отказа проекта.

Если вы не хотите, чтобы ваши проекты потерпели неудачу, найдите время, чтобы сформировать требования и создать дизайн.

Вы не можете сделать их идеальными, но их важность для

эффективного создания хорошего программного обеспечения не следует упускать из виду.

Выявление требований требует активного изучения видения клиента, задавая вопросы о проблемах, которые клиент, возможно, не рассмотрел.

Помимо выявления конкретных потребностей, нужно спрашивать о возможных компромиссах, которые клиент может принять в решении.

С четким представлением о том, что вы пытаетесь выполнить, далее вы можете обратиться к шаблонам дизайна и диаграммам.

Рассмотрим следующий сценарий.

Вас наняли, чтобы спроектировать дом.

Прежде чем приступить к закладке фундамента, вы должны сначала понять, чего хочет домовладелец.

Эта отправная точка известна как выявление требований.

Домовладелец хочет иметь тренажерный зал, санузел, три спальни и гостиную.

Выявление требований подразумевает не только выслушивание того, что говорит вам клиент, но и задавание вопросов для выяснения того, что клиент вам не сказал.

Например, это показалось вам странным, что в этом доме нет кухни?

Это было бы естественным вопросом.

Или все комнаты должны быть одинакового размера?

Насколько большой должен быть дом в целом?

И так далее.

После ответа на эти вопросы у вас теперь есть первоначальный набор требований, позволяющий начать думать о возможных проектах.

Проектная деятельность предполагает принятие требований и определение решения.

Эта деятельность включает в себя создание концептуального дизайна, а затем технического дизайна, что приводит к двум соответствующим видам артефактов, концептуальным макетам и техническим схемам.

Концептуальные макеты представляют то, как будут удовлетворены требования в целом.

На этом этапе вы фокусируетесь на дизайне дома, определяя основные компоненты и их соединения и откладывая технические детали.

И чем яснее концептуальный дизайн, тем лучше будут технические проекты.

После того, как концептуальные макеты завершены, настало время определить технические детали решения.

Из концептуального дизайна вы знаете все основные компоненты и их соединения и обязанности компонентов.

Описание того, как выполняются эти обязанности, является целью технического проектирования.

В техническом дизайне вы начинаете указывать технические детали каждого компонента.

Это делается путем разделения компонентов на более

мелкие компоненты, которые достаточно специфичны для детального проектирования.

Например, компонент тренажерного зала потребует дополнительных компонентов, таких как пол.

Пол будет отвечать за поддержание большого веса.

Домовладелец тренируется как олимпийский атлет.

Разбивая компоненты все больше и больше на дополнительные компоненты, каждый из которых несет определенные обязанности, вы доходите до уровня, где вы можете сделать детальный дизайн конкретного компонента, например, описать, как укрепить пол.

Технические диаграммы выражают, как решать конкретные проблемы, подобные этой.

И при создании приемлемого решения могут возникнуть компромиссы.

Что делать, если укрепление пола в спортзале требует помещения колонн или балок в подвал под тренажерный зал?

И что, если домовладелец также хочет иметь широкое открытое пространство в подвале с хорошей комнатой отдыха?

Иногда могут возникать такие конфликты.

Вам и домовладельцу необходимо будет выработать компромисс в решении.

Если компоненты, их соединения и их обязанности в вашем концептуальном дизайне оказались невозможными в техническом дизайне, или не в состоянии удовлетворить требованиям, вам нужно будет вернуться к вашему концепту-

альному дизайну и переделать его.

Затем технические диаграммы становятся основой для построения предполагаемого решения.

Компоненты, когда они достаточно проработаны, превращаются в коллекции функций, классов или других компонентов.

Эти части представляют собой гораздо более простую проблему, которую разработчики могут реализовывать индивидуально.

Entity объекты соответствуют некоторому реальному объекту.

Boundary объекты - это объекты, которые находятся на границе между системами.

Control объекты отвечают за координацию.

Когда вы разделяете объекты на более мелкие объекты, вы можете обнаружить, что вы будете идентифицировать разные типы объектов.

И обычно определяют три категории объектов.

Во-первых, это Entity объекты.

Entity объекты наиболее знакомы, потому что они соответствуют некоторому реальному объекту.

Если у вас есть объект, представляющий стул в вашем программном обеспечении, то это Entity объект.

Если у вас есть объект, представляющий здание или клиента, это все Entity объекты или сущности.

Как правило, эти объекты знают свои атрибуты.

Они также смогут модифицировать себя и иметь для этого некоторые правила.

Когда вы идентифицируете объекты для включения в ваше программное обеспечение и разбиваете эти объекты на более мелкие объекты, вы сначала получаете Entity объекты.

Другие категории объектов приходят позже, когда вы начнете думать о техническом дизайне программного обеспечения.

Далее, это Boundary объекты.

Граничные объекты Boundary – это объекты, которые находятся на границе между системами.

Это может быть объект, который соприкасается с другой программной системой, например, объект, который получает информацию из Интернета.

Он также может быть объектом, который несет ответственность за отображение информации пользователю и получение его ввода.

Если вы программируете пользовательский интерфейс –

визуальный аспект программного обеспечения – вы, в основном, работаете с граничными объектами.

Любой объект, который имеет дело с другой системой – пользователем, другой программной системой, Интернетом, – можно считать граничным объектом.

Далее, это объекты управления Control.

Control объектами являются объекты, которые отвечают за координацию.

Вы обнаружите объекты управления при попытке деления большого объекта и обнаружите, что было бы полезно иметь объект, который управляет другими объектами.

Организация программного обеспечения с помощью объектов сущностей, граничных объектов и объектов управления позволяет коду быть более гибким, многоразовым и поддерживаемым.

Для программного обеспечения существуют два типа требований.

Это функциональные требования, которые описывают, что система или приложение должны делать.

Например, мультимедийное приложение имеет функциональное требование о возможности загрузки полноразмерного фильма.

Естественно, что разработка программного обеспечения должна четко определять решение для удовлетворения таких требований.

Кроме функциональных требований, есть также нефунк-

циональные требования, которые определяют, насколько хорошо система или приложение делают то, что она делает.

Такие требования могут описывать, насколько хорошо программное обеспечение работает в определенных ситуациях.

Например, мультимедийное приложение может иметь нефункциональные требования для загрузки полноразмерного фильма с определенной скоростью и для воспроизведения такого фильма в пределах определенного размера памяти.

И для решения важны как функциональные, так и нефункциональные требования.

Другой тип нефункциональных требований касается того, насколько хорошо может развиваться код программного обеспечения.

Например, части реализации, возможно, придется поддерживать использование в других подобных программных продуктах.

Кроме того, реализация может потребовать изменения в будущем.

Таким образом, другие качества, которым должно удовлетворять программное обеспечение, могут включать в себя повторное использование, гибкость и ремонтопригодность.

По мере того, как дизайн детализируется и создается реализация, требуемое качество должно проверяться с помощью таких методов, как пересмотры и тесты.

Кроме того, некоторые качества могут быть проверены с помощью обратной связью конечных пользователей.

При разработке программного обеспечения отправной точкой является то, что ваша программная структура должна соответствовать балансу желаемых качеств.

В частности, существует общий компромисс между производительностью и ремонтопригодностью.

Высокопроизводительный код может быть менее понятным и менее модульным, что делает его менее удобным.

Другим компромиссом является безопасность и производительность.

И дополнительные накладные расходы для высокой безопасности могут снизить производительность. И также дополнительный код для обратной совместимости может ухудшить производительность и ремонтопригодность.

Class Name	
Responsibilities	Collaborators

При планировании какого-либо выступления часто используются карточки заметок.

Карточки заметок помогают вам двигаться логически из одной точки разговора в другую.

Было бы неплохо, если бы у нас было что-то похожее, чтобы логически составлять структуру программного обеспечения при формировании его дизайна.

Вы определяете компоненты, соединения и обязанности по некоторым требованиям при формировании концептуального дизайна. Здесь вы формируете свои первоначальные мысли о том, как вы можете удовлетворить требования.

В техническом дизайне эти компоненты и соединения дополнительно уточняются, чтобы придать им технические детали. Это упрощает их реализацию.

Хотя идентификация компонентов, их обязанностей и связей является хорошим первым шагом в разработке программного обеспечения, мы пока не продемонстрировали способ их представления.

И такой метод есть – это использование карточек CRC, где CRC обозначает класс, ответственность, сотрудничество.

Карты CRC помогают организовывать компоненты в классы, определять их обязанности и определять, как они будут сотрудничать друг с другом.

Рассмотрим, например, банкомат.

Вы вставляете свою банковскую карточку в банкомат, и банкомат просит вас ввести PIN-код, удостоверяющий личность для доступа.

После этого вы можете выбрать положить или снять деньги, или проверить свои остатки.

Этот сценарий определяет основные требования к системе.

Это неполный набор требований, но это хороший старт.

Помните, что требования часто являются неполными и дополняются при дальнейшем взаимодействии с вашим клиентом и конечными пользователями.

Следующим шагом будет разработка банкомата.

Но так как мы формируем концептуальный дизайн, ограничиваясь только идентификацией компонентов, их обязанностей и связей, мы можем представить компоненты с помощью нашей новой техники – карт CRC.

И карты CRC используются для записи, упорядочивания и улучшения компонентов в дизайне.

Карта CRC состоит из трех разделов.

В верхней части карты есть имя класса.

Слева – обязанности класса, а справа – список коллабораторов.

Коллабораторы – это другие классы, с которыми класс взаимодействует, чтобы выполнять свои обязанности.

Чтобы отслеживать каждый компонент и его обязанности с помощью CRC-карты, вы помещаете имя компонента в раздел имени класса и обязанности в разделе обязанностей.

До сих пор это довольно просто.

Но как насчет связей?

В разделе «Коллабораторы» вы перечисляете другие компоненты, к которым ваш текущий компонент подключается или взаимодействует, чтобы выполнять свои обязанности.

И карты CRC сами по себе небольшие, поэтому вы не можете много писать в них.

Это заставляет вас продолжать разбивать каждый компонент на более мелкие компоненты и, в конечном итоге, классы, которые достаточно малы для индивидуального описания.

Теперь, когда мы узнали о CRC-картах, давайте использовать их для разработки нашей банковской системы.

Начнем с базового пользовательского компонента.

В этом примере нашим основным пользователем будет

клиент банка.

Мы размещаем клиентов банка в разделе имени класса.

Обязанности банковского клиента включают ввод банковской карточки или выбор операции, такой как депозит, снятие или проверка остатка на счете.

Перечислим их в разделе ответственности CRC-карты.

И мы поместим банкомат в разделе Коллабораторы.

Тоже самое мы можем сделать для банкомата.

И с нашими картами CRC мы можем объединить вместе компоненты для совместной работы.

Например, положите карту клиента CRC слева и карточку CRC банкомата справа.

Когда карты CRC организованы, вы можете имитировать прототип системы.

Теперь, вы можете заметь, что сам банкомат содержит несколько разных компонентов, которые могут быть отдельными классами для программирования.

Например, есть кард-ридер, клавиатура, дисплей и так далее.

Каждый из этих классов, их обязанности и коллабораторы могут быть описаны на их собственных картах.

При встрече с командой разработчиков программного обеспечения вы можете разложить все карты на столе и обсуждать моделирование того, как эти классы работают с другими классами для выполнения своих обязанностей.

И эти симуляции могут выявлять недостатки в дизайне,

и вы можете экспериментировать с альтернативами, вводя соответствующие карты.

Вопросы

Вопрос 1

Что из следующего является желательными характеристиками дизайна программного обеспечения?

Тесная связь

Ремонтопригодность +

Повторное использование +

Гибкость +

Вопрос 2

Определите два результата процесса проектирования.

Концептуальный дизайн +

Реализация кода

Технический дизайн +

План проектирования

Вопрос 3

Вы пишете CRC-карту для компонента банкомата. В каком разделе вы должны поместить «Отслеживание оставшихся денежных средств».

Риски

Класс

Коллабораторы

Обязанности +

Вопрос 4

Что из этого, вероятно, будет частью концептуального дизайна?

Карты CRC +

Абстрактные типы данных

Методы

Макеты +

Вопрос 5

Когда в процессе проектирования вы, скорее всего, будете создавать карты CRC?

Встречи с клиентами

Концептуальный дизайн +

После выпуска программного обеспечения

Технический дизайн

Вопрос 6

Что из следующего является примером нефункциональных требований?

Производительность +

Доступность +

Предназначение

Безопасность +

Вопрос 7

Выберите категории объектов, которые обычно присутствуют в объектно-ориентированном программном обеспечении.

Entity +

Boundary +

tool

Control +

Вопрос 8

Объект, который отвечает за отображение данных пользователю, может быть рассмотрен в какой категории объекта?

representation

boundary +

entity

control

Вопрос 9

Вы планируете класс профессора как часть своего программного обеспечения. Что из следующего вы считаете collaborator?

Отслеживать статус работника

Курс

Студент +

Учебный курс +

Вопрос 10

Что является способом выражения требования в этой форме? «Как ____, я хочу ____, так что ____».

История пользователя +

Концептуальный макет

Абстракция объекта

Ключевое понятие

Задание

Как только возникает требование, оно должно быть выра-

жено в той или иной форме.

Один из способов выражения требования называется историей пользователя.

Пользовательская история – это просто требование, часто с точки зрения конечного пользователя, которое указано на естественном языке.

История пользователя выглядит так:

Как ______, я хочу ______, чтобы ______.

Поместите роль пользователя в первый пробел.

Во втором пробеле укажите цель, которую должна достичь пользовательская роль.

Это приведет к некоторой функции, которую вы хотите реализовать.

После этого укажите причину, по которой пользовательская роль хочет эту цель.

После заполнения пользовательской истории вы можете применить объектно-ориентированное мышление к ней, чтобы обнаружить объекты и, возможно, дополнительные требования!

Вопрос 11

Вы программист, создающий программное обеспечение для банкомата. В какой раздел CRC-карты для компонента банкомата будет включен «Пользователь»?

Коллабораторы +

Обязанности

Объект

Класс

Вопрос 12

Во время концептуального дизайна вы будете говорить о …:

Компромиссах +

Требованиях +

Технических диаграммах

Макетах +

Основные понятия

Объектно-ориентированный подход зародился в программировании в середине прошлого века.

Первым объектно-ориентированным языком был Simula (Simulation of real systems - моделирование реальных систем), разработанный в 1960 году исследователями Норвежского вычислительного центра.

В 1970 году Алан Кей и его исследовательская группа в Xerox PARK создали персональный компьютер Dynabook и первый чистый объектно-ориентированный язык программирования - Smalltalk для программирования Dynabook.

В 1980-х годах Грэди Буч опубликовал документ под названием «Объектно-ориентированный дизайн», в котором в основном был представлен дизайн для языка программирования Ada. В последующих изданиях он расширил свои идеи до полного объектно-ориентированного метода проектирования.

В 1990-х годах Coad включил поведенческие идеи в объектно-ориентированные методы.

Первым объектно-ориентированным языком был Simula (Simulation of real systems – моделирование реальных систем), разработанный в 1960 году исследователями Норвежского вычислительного центра.

В 1970 году Алан Кей и его исследовательская группа в Xerox PARK создали персональный компьютер Dynabook и первый чистый объектно-ориентированный язык програм-

мирования – Smalltalk для программирования Dynabook.

В 1980-х годах Грэди Буч опубликовал документ под названием «Объектно-ориентированный дизайн», в котором в основном был представлен дизайн для языка программирования Ada. В последующих изданиях он расширил свои идеи до полного объектно-ориентированного метода проектирования.

В 1990-х годах Coad включил поведенческие идеи в объектно-ориентированные методы.

Другими значительными нововведениями были методы моделирования объектов Object Modelling Techniques (OMT) Джеймса Рамбо и объектно-ориентированная программная инженерия Object-Oriented Software Engineering (OOSE) Ивара Джекобсона.

С появлением первых компьютеров появились языки программирования низкого уровня.

Языки программирования, ориентированные на конкретный тип процессора, и, операторы которых были близки к машинному коду.

Дальнейшая эволюция языков программирования привела к появлению языков высокого уровня, что позволило отвлечься от системы команд конкретного типа процессора.

При этом происходило смещение от программирования деталей к программированию компонентов, развитие инструментов программирования и возрастание сложности программных систем.

Также развивался подход или стиль написания программ.

В начале использовалось процеду́рное программи́рование, при котором последовательно выполняемые операторы собирались в подпрограммы.

При этом данные и процедуры для их обработки формально не были связаны.

Как следствие возрастания сложности программного обеспечения появилось структу́рное программи́рование – методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков.

И наконец появилось объектно-ориентированное программирование – методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Событи́йно-ориенти́рованное программи́рование – парадигма программирования, в которой выполнение программы определяется событиями – действиями пользователя, сообщениями других программ и потоков, и событиями операционной системы.

Компонентно-ориентированное программирование – парадигма программирования, опирающаяся на понятие компонента – независимого модуля исходного кода программы, предназначенного для повторного использования и развёртывания, и реализующегося в виде множества языковых кон-

струкций.

В 1960-х годах двумя наиболее популярными языками программирования были COBOL и Fortran.

Эти языки следовали императивной парадигме, которая разбивала большие программы на более мелкие программы, называемые подпрограммами, которые похожи на методы в Java.

В 1960-х, время обработки компьютера было дорогостоящим.

Поэтому было важно максимизировать производительность обработки.

Это достигалось за счет наличия глобальных данных, так как они все располагались в одном месте в памяти компьютера для программы.

С глобально доступными переменными все подпрограммы могли получить к ним доступ для выполнения необходимых вычислений.

Однако при этом возникали некоторые проблемы.

С глобальными данными возможно, что изменения в данных могут иметь побочные эффекты для программы.

Иногда подпрограммы запускались с теми данными, которые были не такими, как ожидалось.

Необходимость лучшего управления данными привела к изменениям в императивном программировании и появлению таких языков, как Algol 68 и Pascal в 1970-х годах.

Была введена идея локальных переменных.

Подпрограммы назывались процедурами, которые могут содержать вложенные процедуры.

И каждая процедура могла иметь свои собственные переменные.

Алгол 68 и Паскаль поддерживают понятие абстрактного типа данных, который является типом данных, который определен программистом и не встроен в язык.

Абстрактный тип данных представляет собой, по существу, сгруппированную связанную информацию, которая обозначается типом.

Это был способ организации данных.

Разработчики могли писать свое программное обеспечение с использованием этих типов аналогично встроенным типам языков.

Имея переменные в разных областях видимости, можно было разделить данные на разные процедуры.

Таким образом, процедура могла быть единственной, которая могла модифицировать эту часть данных, позволяя помещать данные в локальную область действия и не беспокоится о том, что они могли изменяться другой процедурой.

По мере того, как время обработки данных компьютерами становилось дешевле, а человеческий труд становился дороже, основным фактором в разработке программного обеспечения стал человеческий фактор.

И задачи становились все более сложными.

Это означало, что программное обеспечение становилось

настолько большим, что, имея только один файл для всей программы, программу становилось трудно поддерживать.

Появились новые языки, такие как C и Modula-2, которые предоставили средства для организации программ и позволяли разработчикам легко создавать несколько уникальных копий своих абстрактных типов данных.

Теперь программы могли быть организованы в отдельные файлы.

В C каждый файл содержал все связанные с ним данные и функции, которые обрабатывали эти данные, и объявлял, к чему можно получить доступ с помощью отдельного файла, называемого файлом заголовка.

Но при этом еще существовали проблемы, которые не решались этими языками программирования.

Эти языки не позволяли абстрактному типу данных наследовать от другого типа данных.

Это означает, что можно было определять столько типов данных, сколько было нужно, но не нельзя было объявить, что один тип является расширением другого типа.

В 1980-х годах, при разработке программного обеспечения стали популярными концепции объектно-ориентированного дизайна, которые являются центральными для объектно-ориентированного программирования.

Цель объектно-ориентированного дизайна состоит в том, чтобы облегчить запись абстрактного типа данных, структурировать систему вокруг абстрактных типов данных, назы-

ваемых классами, и ввести возможность абстрактного типа данных расширять другой, введя понятие, называемое наследованием.

С помощью объектно-ориентированной парадигмы программирования теперь можно было создавать программную систему, состоящую из полностью абстрактных типов данных.

Преимущество этого заключается в том, что система будет имитировать структуру задачи, а это означает, что любая объектно-ориентированная программа способна представить объекты или идеи реального мира с большей точностью.

Файлы классов заменили стандартные файлы в C и Modula-2.

Каждый класс определяет тип со связанными данными и функциями.

Эти функции также известны как методы.

Класс действует как фабрика, создавая отдельные объекты определенного типа.

Это позволяет разделять данные и как ими можно манипулировать в отдельные классы.

Объектно-ориентированное программирование стало преобладающей парадигмой программирования.

Популярные современные языки, такие как Java, C ++ и C #, основаны на объектах.

Объектно-ориентированный анализ (ООА) - это процедура определения требований к программному обеспечению и разработка спецификаций программного обеспечения с точки зрения объектной модели программной системы, которая включает в себя взаимодействующие объекты.

Объектно-ориентированный анализ (ООА) – это процедура определения требований к программному обеспечению и разработка спецификаций программного обеспечения с точки зрения объектной модели программной системы, которая включает в себя взаимодействующие объекты.

Основное различие между объектно-ориентированным анализом и другими формами анализа заключается в том, что в объектно-ориентированном подходе требования организуются вокруг объектов, которые объединяют как данные, так и функции.

Они моделируются по объектам реального мира, с которыми взаимодействует система.

В традиционных методах анализа два аспекта – функции и данные – рассматриваются отдельно.

Основными задачами объектно-ориентированного анализа (ООА) являются:

– Идентификация объектов

– Организация объектов путем создания диаграммы объектной модели

– Определение внутренних объектов или атрибутов объекта

– Определение поведения объектов, т. е. действий объектов

– Описание взаимодействия объектов

Объектно-ориентированный дизайн (OOD) - метод проектирования, охватывающий процесс объектно-ориентированной декомпозиции и обозначение для отображения как логических, так и физических, а также статических и динамических моделей проектируемой системы

Объектно-ориентированный дизайн (OOD) предполагает реализацию концептуальной модели, созданной при объектно-ориентированном анализе.

В OOD концепции модели анализа, которые являются независимыми от технологии, отображаются на классы реализации, идентифицируются ограничения и разрабатываются интерфейсы, что приводит к модели для области решений, то есть подробному описанию того, как система должна быть построена на конкретных технологиях.

Детали реализации обычно включают в себя:

Реструктуризацию данных класса при необходимости,

Реализацию методов, то есть внутренних структур данных и алгоритмов,

Реализацию управления и реализацию ассоциаций.

Объектно-ориентированное программирование (ООП) - метод реализации, в котором программы организованы как совместные коллекции объектов, каждый из которых представляет собой экземпляр некоторого класса и чьи классы являются членами иерархии классов, объединенных через отношения наследования

Объектно-ориентированное программирование (ООП) – это парадигма программирования, основанная на объектах

(имеющих как данные, так и методы), целью которых является использование преимуществ модульности и многоразового использования.

Объекты, которые обычно являются экземплярами классов, используются для взаимодействия друг с другом при разработке компьютерных программ.

Важными чертами объектно-ориентированного программирования являются:

– Подход снизу вверх в разработке программы.

– Программы организованы вокруг объектов, сгруппированных по классам.

– Акцентирование на данных с методами при работе с данными объекта.

– Взаимодействие объектов через функции.

– Повторное использование дизайна путем создания новых классов с помощью добавления функций к существующим классам.

Объектная модель, используемая объектно-ориентированной парадигмой, визуализирует элементы в программном приложении с точки зрения объектов.

И понятия объектов и классов неразрывно связаны между собой и составляют основу объектно-ориентированной парадигмы.

Объект является реальным элементом в объектно-ориентированной среде, который может иметь физическое или концептуальное существование.

Класс представляет собой совокупность объектов, имеющих одни и те же свойства, и которые демонстрируют общее поведение.

Объект является реальным элементом в объектно-ориентированной среде, который может иметь физическое или концептуальное существование.

Физическое существование – это например, клиент, автомобиль и т. д .

Или неосязаемое концептуальное существование – например, проект, процесс и т. д.

Каждый объект имеет идентичность, которая отличает ее от других объектов в системе. И состояние, которое определяет характерные свойства объекта, а также значения свойств, которыми обладает объект. А также поведение, которое представляет внешне видимые действия, выполняемые объектом с точки зрения изменений его состояния.

Класс представляет собой совокупность объектов, имею-

щих одни и те же свойства, и которые демонстрируют общее поведение.

Класс дает схему или описание объектов, которые могут быть созданы из него.

Создание объекта как члена класса называется экземпляром.

Таким образом, объект является экземпляром класса.

Класса состоит из набора атрибутов для объектов, которые должны быть созданы из класса.

Разные объекты класса имеют разные значения атрибутов. И атрибуты часто называются данными экземпляра класса.

И класс состоит из набора операций, которые отображают поведение объектов класса.

Операции также называются функциями или методами.

Инкапсуляция - это процесс связывания как атрибутов, так и методов вместе внутри класса.

Наследование - это механизм, позволяющий создавать новые классы из существующих классов путем расширения и уточнения их возможностей.

Полиморфизм подразумевает использование операций по-разному, в зависимости от того, в каком экземпляре они работают.

Инкапсуляция – это процесс связывания как атрибутов, так и методов вместе внутри класса.

Благодаря инкапсуляции внутренние детали класса могут быть скрыты извне.

И инкапсуляция позволяет доступ к элементам класса извне только через интерфейс, предоставляемый классом.

Как правило, класс разработан таким образом, что его данные (атрибуты) могут быть доступны только через методы класса и изолированы от прямого внешнего доступа.

Этот процесс изоляции данных объекта называется скрытием данных.

Любое приложение требует целого ряда объектов, взаимодействующих между собой. И объекты в системе могут взаимодействовать друг с другом, используя передачу сообщений. И сообщение, проходящее между двумя объектами, как правило, однонаправлено.

Передача сообщений позволяет осуществлять все взаимодействия между объектами.

И передача сообщения по существу включает вызов метода класса.

Наследование – это механизм, позволяющий создавать новые классы из существующих классов путем расширения и уточнения их возможностей.

Существующие классы называются базовыми классами, родительскими классами или суперклассами, а новые клас-

сы называются производными классами, дочерними классами или подклассами.

Подкласс может наследовать атрибуты и методы суперкласса при условии, что суперкласс позволяет это.

Кроме того, подкласс может добавлять свои собственные атрибуты и методы и может модифицировать любой из методов суперкласса.

Наследование определяет отношение «is-a».

Полиморфизм в объектно-ориентированной парадигме подразумевает использование операций по-разному, в зависимости от того, в каком экземпляре они работают.

Полиморфизм позволяет объектам с разными внутренними структурами иметь общий внешний интерфейс.

И полиморфизм особенно эффективен при реализации наследования.

Обобщение и специализация представляют собой иерархию отношений между классами, где подклассы наследуются от суперклассов.

Ссылка представляет собой соединение, через которое объект взаимодействует с другими объектами.

Ассоциация - это группа ссылок, имеющих общую структуру и общее поведение.

Агрегация или композиция - это взаимосвязь между классами, при которой класс может состоять из любой комбинации объектов других классов.

Обобщение и специализация представляют собой иерархию отношений между классами, где подклассы наследуются от суперклассов.

В процессе обобщения общие характеристики классов объединяются для формирования класса на более высоком уровне иерархии, т. е. подклассы объединяются для формирования обобщенного суперкласса.

Специализация – это обратный процесс обобщения.

Здесь отличительные особенности групп объектов используются для формирования специализированных классов из существующих классов.

Можно сказать, что подклассы являются специализированными версиями суперкласса.

Ссылка представляет собой соединение, через которое

объект взаимодействует с другими объектами.

Через ссылку один объект может вызывать методы или перемещаться по другому объекту.

Ссылка изображает взаимосвязь между двумя или более объектами.

Ассоциация – это группа ссылок, имеющих общую структуру и общее поведение.

Ассоциация изображает взаимосвязь между объектами одного или нескольких классов.

И ссылка может быть определена как экземпляр ассоциации.

Степень ассоциации обозначает количество классов, участвующих в соединении. И степень ассоциации может быть унарной, бинарной или тройной.

Унарное отношение связывает объекты одного и того же класса.

Бинарное отношение связывает объекты двух классов.

Тройное отношение связывает объекты трех или более классов.

Мощность бинарной ассоциации обозначает количество экземпляров, участвующих в ассоциации. Существует три типа коэффициента мощности, а именно:

Один-к-одному. Один объект класса A связан с одним объектом класса B.

Один-ко-многим. Один объект класса A связан со многими объектами класса B.

Многие-ко-многим. Объект класса A может быть связан со многими объектами класса B, и, наоборот, объект класса B может быть связан со многими объектами класса A.

Агрегация или композиция – это взаимосвязь между классами, при которой класс может состоять из любой комбинации объектов других классов.

Она позволяет размещать объекты непосредственно внутри тела других классов.

Агрегация называется отношением “part–of” или “has–a”, с возможностью навигации от целого к частям.

Агрегатный объект – это объект, состоящий из одного или нескольких других объектов.

Метод объектно-ориентированного моделирования (OOM) визуализирует вещи в приложении с использованием моделей, организованных вокруг объектов.

И любой подход к разработке программного обеспечения проходит через следующие этапы:

Это анализ, дизайн и реализация.

При объектно-ориентированной разработке программного обеспечения разработчик программного обеспечения идентифицирует и организует приложение с точки зрения объектно-ориентированных концепций до их окончательного представления на любом конкретном языке программирования или программных инструментах.

И основными этапами разработки программного обеспечения с использованием объектно-ориентированной мето-

дологии являются объектно-ориентированный анализ, объектно-ориентированный дизайн и объектно-ориентированная реализация.

На этапе объектно-ориентированного анализа, формулируется проблема, определяются пользовательские требования, а затем модель строится на основе объектов реального мира.

Анализ дает модели то, как должна функционировать желаемая система и как она должна развиваться.

При этом модели не содержат каких-либо деталей реализации, чтобы эти модели могли бы быть поняты и изучены любым экспертом, не являющимся техническим специалистом.

Объектно-ориентированный дизайн включает в себя два основных этапа, а именно: дизайн системы и дизайн объектов.

На этапе системного дизайна разрабатывается полная архитектура желаемой системы.

Система определяется как набор взаимодействующих подсистем, которые, в свою очередь, состоят из иерархии взаимодействующих объектов, сгруппированных по классам.

Конструирование системы выполняется на основе как модели анализа, так и предлагаемой архитектуры системы.

Здесь акцент делается на объектах, входящих в систему, а не на процессы в системе.

На этапе дизайна объектов разрабатывается модель на основе как моделей, разработанных на этапе анализа, так и архитектуры, разработанной на этапе дизайна системы. При этом определяются все необходимые классы.

Устанавливаются ассоциации между классами и определяются иерархии классов.

На этапе объектно-ориентированной реализации и тестирования модель дизайна, разработанная при дизайне объектов, преобразуется в код на соответствующем языке программирования.

Создаются базы данных и определяются конкретные требования к оборудованию.

После того, как создается код, он проверяется с использованием специализированных методов для выявления и устранения ошибок в коде.

Принципы объектно-ориентированных систем

Абстракция
Инкапсуляция
Модульность
Иерархия

Типизация
Параллельность
Сохраняемость

Концептуальная структура объектно-ориентированных систем основана на объектной модели.

И объектно-ориентированная система основывается на двух категориях свойств.

Это основные свойства, которые объектно-ориентированная система обязана иметь:

– Абстракция.
– Инкапсуляция.
– Модульность.
– Иерархия.

И дополнительные свойства, которые полезны, но не являются неотъемлемой частью объектной модели:

– Типизация.
– Параллельность.

– Сохраняемость.

Абстракция обозначает существенные характеристики объекта, которые отличают его от всех других видов объектов и, таким образом, обеспечивают четко определенные концептуальные границы относительно перспективы зрителя.

Инкапсуляция - это процесс связывания как атрибутов, так и методов вместе внутри класса.

Модульность - это свойство системы, которая была разложена на множество когезионных и слабо связанных модулей.

Иерархия - это ранжирование или упорядочение абстракции.

Абстракция означает сосредоточиться на существенных особенностях элемента или объекта, игнорируя его посторонние или случайные свойства.

И основные свойства относятся к контексту, в котором используется объект.

Инкапсуляция – это процесс связывания как атрибутов, так и методов вместе внутри класса.

Благодаря инкапсуляции внутренние детали класса могут быть скрыты извне.

Класс имеет методы, которые предоставляют пользовательские интерфейсы, с помощью которых могут использоваться службы, предоставляемые классом.

Модульность – это процесс разложения задачи (программы) на набор модулей, чтобы уменьшить общую сложность проблемы.

И модульность связана с инкапсуляцией.

Модульность может быть визуализирована как способ отображения инкапсулированных абстракций в реальные физические модули, имеющие высокую степень сцепления внутри модулей, а их межмодульное взаимодействие или связь является слабой.

Иерархия – это ранжирование или упорядочение абстракции.

Через иерархию система может состоять из взаимосвязанных подсистем, которые могут иметь свои собственные подсистемы и т. д.

До тех пор, пока не будут достигнуты наименьшие компоненты уровня.

Иерархия использует принцип «разделяй и властвуй».

И иерархия позволяет повторно использовать код.

Двумя типами иерархий являются:

Иерархия «IS-A». Она определяет иерархическую взаимосвязь в наследовании, в которой из суперкласса может быть выведено несколько подклассов, которые могут снова иметь подклассы и т. д.

И иерархия «PART-OF» – определяет иерархическую взаимосвязь в агрегации, посредством которой класс может состоять из других классов.

Тип является характеристикой набора элементов.

Типизация - это применение понятия о том, что объект является экземпляром одного класса или типа.

Параллельность позволяет одновременно выполнять несколько задач или процессов.

Сохраняемость — свойство объекта непрерывно сохранять требуемые эксплуатационные показатели в течение (и после) срока хранения и транспортирования.

Согласно теории абстрактного типа данных, тип является характеристикой набора элементов.

В ООП класс визуализируется как тип, имеющий свойства, отличные от любых других типов.

Типизация – это применение понятия о том, что объект является экземпляром одного класса или типа.

Типизация также предусматривает, что объекты разных типов обычно не являются взаимозаменяемыми; и могут быть взаимозаменяемы только в очень ограниченном порядке, если это абсолютно необходимо.

Два типа типизации – это строгая типизация – здесь операция над объектом проверяется во время компиляции.

И слабая типизация – здесь сообщения могут быть от-

правлены в любой класс.

Операция проверяется только во время выполнения.

Параллельность в операционных системах позволяет одновременно выполнять несколько задач или процессов.

Большинство систем имеют несколько потоков, при этом некоторые активные, а некоторые ждут процессор, некоторые приостановлены и некоторые завершены.

Системы с несколькими процессорами допускают одновременные потоки управления; но системы, работающие на одном процессоре, используют соответствующие алгоритмы для обеспечения равного времени процессора для потоков, чтобы обеспечить параллелизм.

В объектно-ориентированной среде существуют активные и неактивные объекты.

Активные объекты имеют независимые потоки управления, которые могут выполняться одновременно с потоками других объектов.

И активные объекты синхронизируются друг с другом, а также с чисто последовательными объектами.

Объект занимает пространство памяти и существует в течение определенного периода времени.

В традиционном программировании продолжительность жизни объекта обычно была продолжительностью выполнения программы, которая ее создала.

В файлах или базах данных продолжительность жизни объекта больше, чем продолжительность процесса, создаю-

щего объект.

Свойство, с помощью которого объект продолжает существовать даже после того, как его создатель перестает существовать, известно, как сохраняемость.

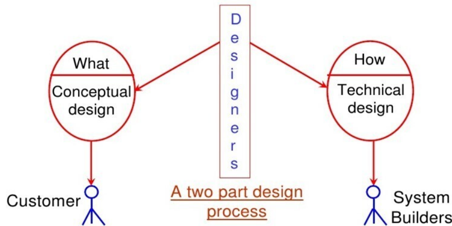
Принципы ООД (Объектно-ориентированного дизайна)

Если вы хотите построить дом, вы не забьёте и гвоздя без проекта.

Аналогичным образом, для решения сложной задачи с помощью программного обеспечения вы не погружаетесь прямо в кодирование.

Вам нужен концептуальный дизайн, чтобы разложить задачу на управляемые части.

И вам также нужен технический дизайн для описания решения, чтобы оно было достаточно понятно разработчикам программного обеспечения.



На протяжении многих лет люди пробовали множество подходов для упрощения проектирования.

Например, существуют стратегии проектирования, подходящие для решения определенных задач.

Если у вас есть задача, связанная с обработкой данных, возможно, вы используете программирование сверху вниз.

Эта стратегия отображает процессы обработки данных в задаче на вызовы процедур.

По мере того, как вы раскладываете необходимые процессы обработки сверху вниз, вы создаете дерево процедур для возможного решения.

И эти процедуры реализуются на определенном языке программирования.

Для многих видов сложных задач имеет смысл подумать

о концепциях, использующих объекты.

Например, любое существительное в описании задачи может быть важным объектом.

Реальный мир, где возникают задачи, наполнен объектами. И это привело к популярности объектно-ориентированного программирования.

Но даже здесь вы все равно не переходите сразу от задачи к написанию кода.

Существует концептуальный дизайн, включающий в себя объектно-ориентированный анализ для идентификации ключевых объектов в задаче.

Существует также технический дизайн, включающий в себя объектно-ориентированный дизайн для дальнейшего уточнения деталей объектов, включая их атрибуты и поведение.

Проектная деятельность происходит итеративно и непрерывно.

Целью дизайна программного обеспечения является построение и доработка моделей всех объектов.

И эти модели полезны на протяжении всего процесса проектирования.

Первоначально основное внимание должно быть сосредоточено на объектах сущностей entity из пространства задачи.

По мере появления решения вы вводите объекты управления control, которые принимают события и координируют действия. Вы также вводите пограничные объекты boundary,

которые подключаются к службам вне вашей системы.

Модели часто выражаются визуально с помощью унифицированного языка моделирования или UML.

В объектно-ориентированном моделировании у вас есть разные типы моделей или диаграмм UML, чтобы сосредоточиться на различных аспектах программного обеспечения, например, структурная модель, для описания того, что делают объекты и как они связаны.

И структурная модель похожа на масштабную модель здания для понимания пространственных отношений.

Чтобы справиться со сложностью задачи, вы можете применять принципы дизайна для упрощения объектов.

Например, разделить их на более мелкие части и посмотреть на общие черты, которые можно обрабатывать последовательно.

Также необходим постоянный пересмотр и оценка моделей для обеспечения того, чтобы дизайн соответствовал задаче и отвечал целям задачи.

Модели также служат в качестве проектной документации для программного обеспечения и могут быть легко сопоставлены с скелетным исходным кодом, особенно для объектно-ориентированного языка, такого как Java.

И это может послужить хорошим началом для разработчиков, реализующих программное обеспечение.

Когда вы разрабатываете объектно-ориентированные программы, вы создаете модели того, как объекты представ-

лены в вашей системе. Эти модели не могут быть разработаны без реализации определенных принципов.

Для того чтобы система была объектно-ориентированной, она должна придерживаться определенных принципов проектирования.

И один из принципов проектирования в объектно-ориентированном моделировании, – это абстракция.

Абстракция - это идея упрощения концепции в области задачи
до ее сути в определенном контексте.

Абстракция – один из основных способов, с помощью которых люди справляются со сложностью задачи.

Абстракция – это идея упрощения концепции в области задачи до ее сути в определенном контексте.

Абстракция позволяет лучше понять концепцию, разбив ее на упрощенное описание, которое игнорирует несуще-

ственные детали.

Абстракция концентрируется на внешних характеристиках объекта и позволяет отделить наиболее существенные особенности его поведения от менее существенных.

И граница между существенными и несущественными деталями с точки зрения дизайна называется барьером абстракции.

И задачей дизайна является выделение полного и достаточного набора абстракций.

Например, мы могли бы создать абстракцию для еды.

В контексте здоровья ее пищевая ценность, а не ее стоимость, будет частью упрощенного описания пищи.

Хорошая абстракция подчеркивает основы, необходимые для концепции, и устраняет детали, которые не являются существенными.

Также абстракция концепции должна иметь смысл для цели концепции.

Эта идея применяет правило наименьшего удивления.

То есть абстракция фиксирует основные атрибуты и поведение для концепции без каких-либо сюрпризов и не содержит определений, выходящих за рамки ее возможностей.

Вы не хотите удивить любого, кто пытается понять вашу абстракцию с нерелевантными характеристиками.

В объектно-ориентированном моделировании абстракция относится непосредственно к понятию класса.

Когда вы используете абстракцию для определения основ-

ных характеристик для какой-либо концепции, имеет смысл определить все эти детали в классе, названном соответственно концепции.

Класс похож на шаблон для экземпляров концепции.

Объект, созданный из класса, затем имеет существенные детали для представления экземпляра некоторого понятия.

Позже мы подробно рассмотрим, как формировать классы, используя абстракцию.

Давайте возьмем понятие человека. Каковы основные характеристики человека?

Это, трудно сказать, потому что человек настолько расплывчатое понятие, и мы не сказали, какова цель нашего человека.

Абстракции, которые вы создаете, относятся к некоторому контексту, и для одной концепции могут быть разные абстракции.

Например, если вы создаете приложение для вождения, вы должны описать человека в контексте водителя.

В другом примере, если вы создаете приложение для ресторана, тогда вы должны описывать человека в контексте клиента.

Вам решать какую выбрать абстракцию, наиболее подходящую для вашей цели.

Прежде чем мы начнем создавать абстракцию, нам нужен контекст для нее.

Контекст имеет решающее значение при формировании

абстракции.

После определения контекста и абстракции, мы определяет характеристики или атрибуты абстракции.

И в дополнение к атрибутам абстракция должна описывать базовое поведение концепции.

Всякий раз, когда мы создаем абстракцию, нам нужно помнить о контексте.

Если контекст изменяется, тогда может измениться и абстракция. А затем могут измениться ее атрибуты и поведение.

Инкапсуляция формирует автономный объект путем связывания данных и функций, которые он требует для работы, предоставляет интерфейс, посредством которого другие объекты могут обращаться к нему и использовать его, и ограничивает доступ к некоторым внутренним деталям.

Инкапсуляция является фундаментальным принципом в объектно-ориентированном моделировании и программировании.

Есть много вещей, которые вы можете представить, как объекты.

Например, вы можете представить курс как объект.

Объект курса может иметь значения атрибутов, такие как определенное количество учащихся, стоимость и предварительные условия, а также конкретные поведения, связанные с этими значениями атрибутов.

И класс курса определяет основные атрибуты и поведение всех объектов курса.

Инкапсуляция включает в себя три идеи.

Как следует из названия, речь идет о создании своего рода капсулы. Капсула содержит что-то внутри.

И некоторое из этого что-то вы можете получить снаружи, а некоторое вы не можете.

Во-первых, вы объединяете значения атрибутов или данные, а также поведение или функции, которые совместно используют эти значения в автономном объекте.

Во-вторых, вы можете выставить наружу определенные данные и функции этого объекта, к которым можно получить доступ из других объектов.

В-третьих, вы можете ограничить доступ к определенным данным и функциям только внутри этого объекта.

Короче говоря, инкапсуляция формирует автономный объект путем связывания данных и функций, которые он требует для работы, предоставляет интерфейс, посредством которого другие объекты могут обращаться к нему и исполь-

зовать его, и ограничивает доступ к некоторым внутренним деталям.

И вы определяете класс для данного типа объекта.

Абстракция помогает определить, какие атрибуты и поведение имеют отношение к концепции в некотором контексте.

Инкапсуляция гарантирует, что эти характеристики объединены вместе в одном классе.

Отдельные объекты, созданные таким образом из определенного класса, будут иметь свои собственные значения данных для атрибутов и будут демонстрировать результат поведения.

Вы обнаружите, что программирование проще, когда данные и код, который управляет этими данными, расположены в одном месте.

Данные объекта должны содержать только то, что подходит для этого объекта.

Помимо атрибутов, класс также определяет поведение через методы.

Для объекта класса методы управляют значениями атрибутов или данными в объекте для достижения фактического поведения.

Вы можете предоставить определенные методы для доступа объектам других классов, таким образом, предоставляя интерфейс для использования класса.

И инкапсуляция помогает с целостностью данных.

Вы можете определить определенные атрибуты и методы

класса, которые должны быть ограничены извне для доступа.

И на практике вы часто представляете внешний доступ ко всем атрибутам через определенные методы.

Таким образом, значения атрибутов объекта не могут быть изменены непосредственно через назначения переменных.

В противном случае такие изменения могут нарушить некоторое допущение или зависимость для данных внутри объекта.

Кроме того, инкапсуляция может обеспечить конфиденциальность информации.

Например, вы можете разрешить классу студента сохранять среднюю оценку баллов.

Сам класс студента может поддерживать запросы, связанные со средней оценкой баллов, но без необходимости показывать фактическое значение баллов.

Инкапсуляция помогает с изменениями программного обеспечения.

Доступный интерфейс класса может оставаться неизменным, а реализация атрибутов и методов может измениться.

Пользователям, использующим класс, не нужно заботиться о том, как реализация фактически работает за интерфейсом.

В программировании такого рода подход обычно называют черным ящиком.

Подумайте о классе, как о черном ящике, который вы не

можете видеть внутри, для получения подробной информации о том, как представлены атрибуты или как методы вычисляют результат, но вы предоставляете входные данные и получаете результаты посредством вызова методов.

Так как внутренняя работа не имеет отношения к внешнему миру, это обеспечивает абстракцию, которая эффективно снижает сложность для пользователей класса.

И это увеличивает повторное использование, потому что другому классу нужно знать только правильный метод вызова, чтобы получить желаемое поведение, какие аргументы поставлять в качестве входных данных и что будет отображаться как результат.

Инкапсуляция является ключевым принципом разработки в хорошо написанной программе.

Она поддерживает модульность и простоту работы с программным обеспечением.

Декомпозиция – это разделение целого на разные части и объединение отдельных частей с различными функциональными возможностями вместе, чтобы сформировать целое.

Декомпозиция берет целую вещь и делит ее на разные части.

Или, с другой стороны, берет кучу отдельных частей с различными функциональными возможностями и объединяет их вместе, чтобы сформировать целое.

Разложение позволяет вам разложить проблему на части, которые легче понять и решить.

Разделяя вещь на разные части, вы можете более легко разделить обязанности этой вещи.

Общее правило для разложения состоит в том, чтобы посмотреть на разные обязанности чего-то целого и оценить, как вы можете разделить это целое на разные части, каждую со своей конкретной обязанностью.

Это связывает целое с несколькими различными частями.

Иногда целое делегирует конкретные обязанности своим частям.

Например, холодильник делегирует замораживание пищи и хранение этой пищи в морозильной камере.

Так как разложение позволяет создавать четко определенные части, вполне естественно, что эти части являются отдельными.

Целое может иметь фиксированное или динамическое число частей определенного типа.

Если существует фиксированное число, то за время жизни всего объекта он будет иметь именно это количество объектов частей.

Например, холодильник имеет фиксированное количество морозильников, только один.

Это не меняется со временем, но иногда есть части с динамическим числом.

Объект может получить новые экземпляры объектов частей за время его существования.

Например, холодильник может иметь динамическое количество полок с течением времени.

И сама часть может также служить целым, содержащим дополнительные составные части.

В декомпозиции играет роль время жизни всего объекта, а также время жизни объектов частей и то, как они могут соотноситься между собой.

Например, холодильник и морозильник имеют одинако-

вый срок службы.

И одно не может существовать без другого.

Если вы откажетесь от холодильника, вы также избавитесь от морозильной камеры.

Но срок жизни также может быть не связан.

У холодильника и продуктов питания разные сроки службы. И каждый может существовать независимо.

Также вы можете иметь целые вещи, которые имеют общие части в одно и то же время.

Например, человек, у которого есть дочь в одной семье, а также супруга в другой семье.

Эти две семьи считаются отдельными целыми, но они одновременно имеют одну и ту же общую часть.

Однако иногда совместное использование невозможно.

Например, пищевой продукт в холодильнике не может одновременно находиться в духовке.

В целом, разложение помогает разбить задачу на более мелкие части.

И сложная вещь может быть составлена из отдельных более простых частей.

И важным является понимания – это то, как части относятся к целому, фиксированное или динамическое их число, их время жизни и совместное использование.

Обобщение помогает сократить избыточность при решении задачи.

Идея объектно-ориентированного моделирования и программирования заключается в создании компьютерного представления концепций в пространстве задачи.

И принцип проектирования, называемый обобщением, помогает сократить избыточность при решении задачи.

Многие виды поведения в реальном мире действуют посредством повторяющихся действий.

И мы можем моделировать поведение с помощью методов.

Это позволяет нам обобщать поведение и устраняет необходимость иметь идентичный код, разбросанный во всей программе.

Например, возьмите код создания и инициализации массива.

Мы можем обобщить этот повторяющийся код, сделав отдельный метод. Это помогает нам уменьшить количество почти идентичного кода в нашей системе.

Методы – это способ применения одного и того же поведения к другому набору данных.

Обобщение часто используется при реализации алгоритмов, которые предназначены для выполнения одного и того же действия на разных наборах данных.

Мы можем обобщать действия в метод и просто передавать другой набор данных через аргументы.

Так где же мы можем применить обобщение?

Если мы можем повторно использовать код внутри метода и метод внутри класса, то можем ли мы повторно использовать код класса?

Можем ли мы обобщить классы?

Обобщение является одним из основных принципов объектно-ориентированного моделирования и программирования.

Но здесь обобщение достигается иначе, чем обобщение с помощью методов.

Обобщение в ООП может быть выполнено классами через наследование.

В обобщении мы принимаем повторяющиеся, общие характеристики двух или более классов и переносим их в другой класс.

В частности, вы можете иметь два класса, родительский

класс и дочерний класс.

Когда дочерний класс наследуется от родительского класса, дочерний класс будет иметь атрибуты и поведение родительского класса.

Вы размещаете общие атрибуты и поведение в своем родительском классе.

Может быть несколько дочерних классов, которые наследуются от родительского класса, и все они получат эти общие атрибуты и поведение.

У дочерних классов также могут быть дополнительные атрибуты и поведение, которые позволяют им быть более специализированными в том, что они могут делать.

В стандартной терминологии родительский класс известен как суперкласс, а дочерний класс называется подклассом.

Одно из преимуществ такого обобщения заключается в том, что любые изменения кода, которые являются общими для обоих подклассов, могут быть сделаны только в суперклассе.

Второе преимущество заключается в том, что мы можем легко добавить больше подклассов в нашу систему, не выписывая для них все общие атрибуты и поведение.

Через наследование все подклассы класса будут обладать атрибутами и поведением суперкласса.

Наследование и методы иллюстрируют принцип обобщения в проектировании.

Мы можем писать программы, способные выполнять одни и те же задачи, но с меньшим количеством кода.

Это делает код более многоразовым, потому что разные классы или методы могут совместно использовать одни и те же блоки кода.

Системы упрощаются, потому что у нас нет повторяющегося кода.

Обобщение помогает создать программное обеспечение, которое будет легче расширять, проще применять изменения и упрощает его поддержку.

Обобщение и наследование являются одной из наиболее сложных тем в объектно-ориентированном программировании и моделировании.

Наследование – это мощный инструмент проектирования, который помогает создавать понятные, многоразовые и поддерживаемые программные системы.

Однако неправильное наследование может привести к плохому коду.

Это происходит, когда принципы проектирования используются ненадлежащим образом, создавая больше проблем, хотя они предназначены для их решения.

Итак, как мы можем понять, злоупотребляем ли мы наследованием?

Есть несколько моментов, о которых нужно знать, когда рассматривается наследование.

Во-первых, вам нужно спросить себя, пользуюсь ли я на-

следованием, чтобы просто использовать общие атрибуты или поведение, не добавляя ничего особенного в подклассы?

Если ответ «да», тогда вы неправильно используете наследование.

Это является признаком неправильного использования, потому что нет никаких оснований для существования подклассов, так как суперкласса уже достаточно.

Скажем, вы проектируете ресторан пиццы. И вам нужно смоделировать все различные варианты пиццы, которые есть у ресторана в меню.

Учитывая различную комбинацию начинок и названий, которые вы можете использовать для пиццы, может возникнуть соблазн разработать систему, использующую наследование.

```
public class Pepperoni extends Pizza {

    public Pepperoni(String size,
    String crust) {
        super(size, crust);
        super.addTopping("pepperoni");
    }
}
```

И класс пиццы может обобщен.

Это кажется разумным, но давайте посмотрим, почему это является неправильным использованием наследования.

Несмотря на то, что пицца pepperoni – более специфическая пицца, она не очень отличается от суперкласса.

Вы можете видеть, что конструктор pepperoni использует конструктор пиццы и добавляет начинки, используя метод суперкласса.

В этом случае нет причин для использования наследования, потому что вы можете просто использовать только класс пиццы для создания пиццы с пепперони в качестве верхней части.

Второй признак ненадлежащего использования обобщения – если вы нарушаете Принцип Замещения Лискова.

Принцип гласит, что подкласс может заменить суперкласс, тогда и только тогда, когда подкласс не изменяет функциональность суперкласса.

Как этот принцип может быть нарушен через наследование?

```
public class Animal {
    private int numberOfLegs;
    private boolean hasTail;
    /* Other characteristics of an animal can be included here */

    public Animal(int legs, boolean tail) {
        this.numberOfLegs = legs;
        this.hasTail = tail;
    }

    public void walk() { … }
    public void run() { … }
    public void eat() { … }
    /* Other behaviors of an animal can be included here */
}
```

Давайте посмотрим на этот пример.

Это наш обобщенный класс животных, и он знает, как есть, гулять и бегать.

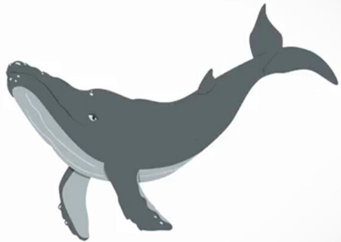
Теперь, как мы можем ввести подкласс, который нарушит принцип замещения Лискова?

```java
public class Whale extends Animal {
    public Whale () {
        super(0, true);
    }

    private void swim() { … }

    public void run() {
        this.swim();
    }

    public void walk() {
        this.swim();
    }
}
```



Что, если у нас есть этот тип животных?

Кит не знает, как гулять и бегать.

Гулять и бегать – это поведение наземных животных.

И принцип замещения Лискова здесь нарушен, потому что класс китов переопределяет класс животных, и ходячие функции заменяет на плавательные функции.

Пример плохого наследования можно увидеть и в библиотеке коллекций Java.

Вы когда-нибудь использовали класс стека в Java?

Стек имеет небольшое количество четко определенных поведений, таких как peak, pop и push.

Но класс стека наследуется от класса вектора.

Это означает, что класс стека может возвращать элемент по указанному индексу, извлекать индекс элемента и даже

вставлять элемент по определенному индексу.

И это не является поведением стека, но из-за плохого использования наследования это поведение разрешено.

Если наследование не соответствует вашим потребностям, подумайте, подходит ли декомпозиция.

Смартфон – это хороший пример того, где декомпозиция работает лучше, чем наследование.

Смартфон имеет характеристики телефона и камеры.

И для нас не имеет смысла наследовать от телефона, а затем добавлять методы камеры в подкласс смартфон.

Здесь мы должны использовать декомпозицию для извлечения ответственностей камеры и размещения их в классе смартфона.

Тогда смартфон будет косвенно обеспечивать функциональность камеры в телефоне.

Наследование может быть сложным принципом разработки, но это очень мощный метод.

Помните, что общая цель заключается в создании многоразовых, гибких и поддерживаемых систем.

И наследование – это просто один из способов помочь вам достичь этой цели.

И важно понимать, что этот метод полезен только при правильном использовании.

Принцип Абстракции в UML

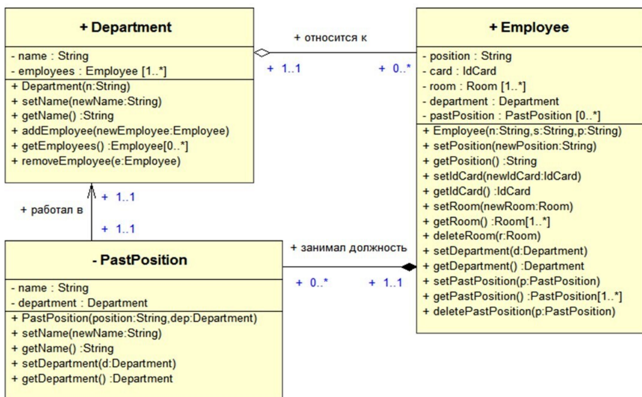
При проектировании здания архитекторы создают эскизы, чтобы визуализировать и экспериментировать с различными проектами.

Эскизы быстро создаются и интуитивно понятны для представления дизайна клиенту, но эти эскизы недостаточно подробны для строителей.

Когда архитекторы общаются с людьми, которые будут строить здание, они предоставляют подробные чертежи, которые содержат точные измерения различных компонентов.

Эти дополнительные детали позволяют строителям точно построить то, что предлагает архитектор.

Для программного обеспечения, разработчики используют технические диаграммы, называемые UML диаграммами, для выражения своих проектов.



Напомним, что для концептуального дизайна мы использовали CRC-карточки, которые аналогичны эскизам архитекторов для визуализации и экспериментов с различными проектами.

Карточки CRC хороши только для прототипирования и моделирования проектов на более высоком уровне абстракции.

Однако, для реализации, нужна техника, которая больше похожа на план.

Диаграммы классов UML позволяют представить дизайн более подробно, чем карточки CRC, но это представление будет все еще визуальным.

Диаграммы классов намного ближе к реализации и могут быть легко преобразованы в классы в коде.

Принцип абстракции дизайна представляет собой идею упрощения концепции в области задачи до ее сути в каком-то контексте.

Абстракция позволяет лучше понять концепцию, разбив ее на упрощенное описание, которое игнорирует несущественные детали.

Вы можете сначала применить абстракцию на уровне дизайна, используя диаграммы классов UML, а затем преобразовать дизайн в код.

Итак, как например, класс продуктов питания выглядел бы в диаграмме классов?

Это представление диаграммы класса продуктов питания.

Каждый класс в диаграмме классов представлен полем.

И каждая диаграмма разделена на три секции, как в CRC-карточке.

Верхняя часть – это имя класса.

Средняя часть – это раздел свойств.

И это эквивалентно переменным-членам в классе Java, и эта часть определяет атрибуты абстракции.

И, наконец, нижняя часть – это раздел операций, который эквивалентен методам в классе Java и определяет поведение абстракции.

Свойства, которые эквивалентны переменным-членам Java, состоят из имени переменной и типа переменной.

Типы переменной, как и в Java, могут быть классами или примитивными типами.

Операции, эквивалентные методам Java, состоят из имени операции, списка параметров и типа возвращаемого значения.

Теперь, если мы сравним карточку CRC с нашей диаграммой классов, вы можете заметить, как некоторые из обязанностей карточки превратились в свойства в диаграмме классов.

Некоторые из них стали операцией.

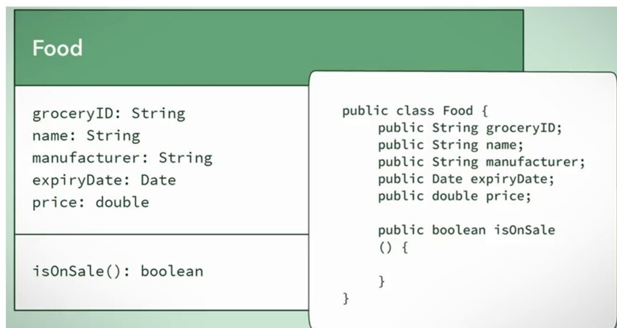
Конечно, вы можете использовать CRC-карточки для абстрагирования объекта, но тут возникают двусмысленности, которые препятствуют программисту перевести CRC-карточку в код.

Одна из двусмысленностей заключается в том, что CRC-карточка не показывает разделения между свойствами и операциями.

Все они перечислены вместе.

Теперь, когда у нас есть представление диаграммы клас-

сов, давайте реализуем его в код Java.



Диаграммы классов очень близки к реализации, что делает перевод на Java очень простым.

Имя класса в диаграмме превращается в класс в Java.

Свойства в диаграмме классов превращаются в переменные-члены.

И, наконец, операции превращаются в методы.

Преобразование кода в диаграмму классов также является простым.

Несмотря на дополнительные подробности, которые может предоставить диаграмма классов, CRC-карточки успешно используются для имитации и прототипирования различных конструкций.

А тот факт, что они далеки от кода, заставляет вас сосредоточиться на задаче, а не на реализации.

С другой стороны, диаграммы классов намного ближе к коду, и вы можете четко передать свой технический дизайн разработчикам.

Но поскольку вам нужно указать специфичные для кода вещи, такие как списки параметров и возвращаемые значения, диаграммы классов слишком детализированы для концептуального дизайна.

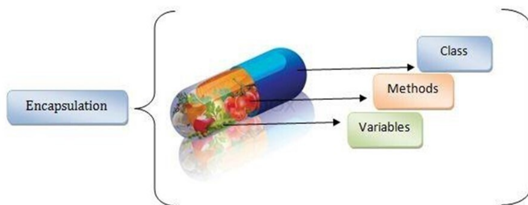
Детали будут отвлекать и отнимать много времени, при создании первоначальных проектов.

Принцип Инкапсуляции в UML

Теперь, когда вы понимаете основные принципы объектно-ориентированного программирования, вам нужно научиться их применять.

Давайте посмотрим, как применить инкапсуляцию.

Как вы помните, инкапсуляция включает в себя три идеи.



Во-первых, вы объединяете данные и функции, которые управляют данными, в автономный объект.

Во-вторых, вы можете предоставить определенные данные и функции этого объекта, чтобы к ним можно получить

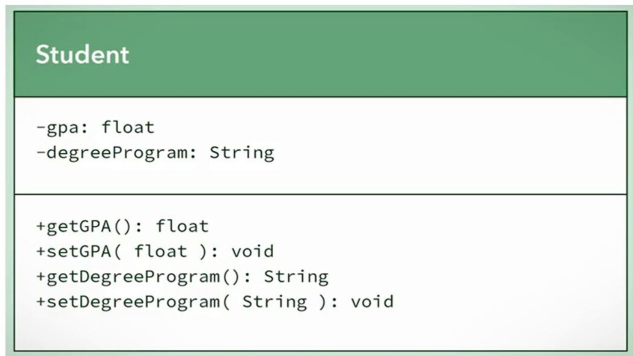
доступ из других объектов.

И в-третьих, вы можете ограничить доступ к определенным данным и функциям только внутри этого объекта.

Итак, как это выглядит в коде?

И как выглядит в дизайне?

Прежде чем перейти к написанию кода, давайте посмотрим на некоторые обозначения в диаграмме классов UML, которые выражают инкапсуляцию.



Если вы создаете систему, которая моделирует студента с использованием инкапсуляции, вы должны иметь все соответствующие данные, определенные в атрибутах класса студента.

Вам также понадобятся публичные методы, которые будут

обращаться к атрибутам.

В этом примере соответствующие данные студента могут быть его программой обучения и баллами.

Класс студента имеет свои атрибуты, скрытые извне.

И это обозначается знаками минуса перед атрибутами.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «ЛитРес».

Прочитайте эту книгу целиком, купив полную легальную версию на ЛитРес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.