

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**

«Автор»

2022

Машнин Т.

Введение в объектно-ориентированный дизайн с Java /
Т. Машнин — «Автор», 2022

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	6
Вопросы	14
Основные понятия	16
Принципы ООД (Объектно-ориентированного дизайна)	28
Принцип Абстракции в UML	39
Принцип Инкапсуляции в UML	43
Принцип Декомпозиции в UML	46
Принцип Обобщения в UML	52
Вопросы	59
Связанность и когезия	63
Разделение ответственостей	67
Скрытие информации	70
Концептуальная целостность	72
Моделирование поведения. UML диаграммы последовательности	74
Задание	78
UML диаграмма состояний	80
Задание	83
Вопросы	84
Паттерны проектирования	87
Factory Method Pattern	92
Abstract Factory Pattern	100
Singleton Pattern	104
Prototype Pattern	107
Builder Design Pattern	109
Structural design patterns. Adapter Pattern	111
Bridge Pattern	115
Composite Pattern	118
Decorator Pattern	126
Facade Pattern	135
Flyweight Pattern	141
Proxy Pattern	144
Задание	148
Задание	150
Вопросы	155
Поведенческие шаблоны проектирования. Chain Of Responsibility Pattern	158
Command Pattern	162
Interpreter Pattern	168
Iterator Pattern	170
Mediator Pattern	172
Memento Pattern	174
Observer Pattern	176
State Pattern	181
Strategy Pattern	188
Template Pattern	190
Visitor Pattern	195

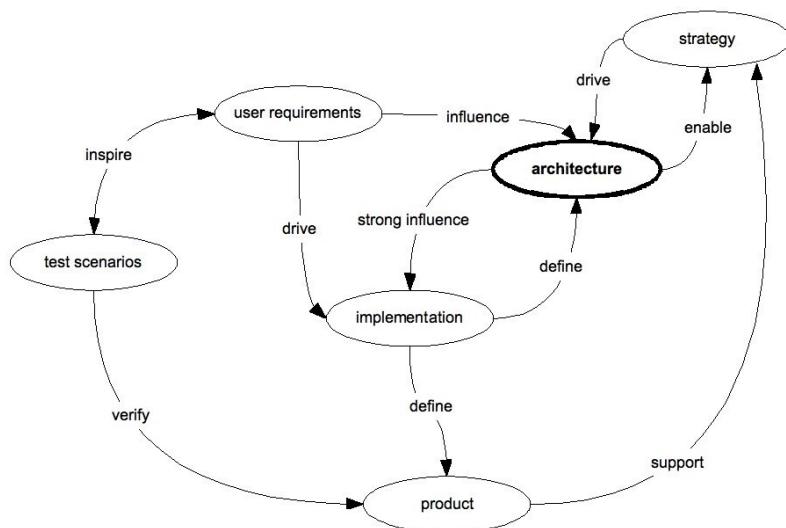
Задание	197
Вопросы	199
MVC Pattern	204
Задание	209
Принципы проектирования. Принцип подстановки Лисков	210
Открыто-закрытый принцип	214
Инверсии зависимостей	217
Принцип композиции объектов	222
Разделение интерфейса	225
Принцип наименьшего знания	227
Анти-паттерны	232
Вопросы	243

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 взаимодействий 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 программируют 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 концептуальному 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.

Границочные 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 нефункциональные 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 дополнитель но 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 коллaborаторов.

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 заметить, 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 коллеги 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

Коллaborаторы

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 выражено 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»?

Коллaborаторы +

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 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.

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 процедурное программирование, 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 структурное программирование – 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.

Событийно-ориентированное программирование – 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, предназначенному для повторного использования и разворачивания, и реализующегося в виде множества языковых конструкций.

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 С 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 С 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, называемых 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 С 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 какими 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 неосозаемое 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 внешние видимые действия, выполняемые объектом с точки зрения изменений его состояния.

Класс представляет собой совокупность объектов, имеющих 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, одностороннее.

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 классы 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 А связан с одним объектом класса В.

Один-ко-многим. Один объект класса А связан со многими объектами класса В.

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

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 (ООМ) 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 методологии 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 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 взаимозаменены 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 отправлены 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, создающего объект.

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

Принципы ООД (Объектно-ориентированного дизайна)

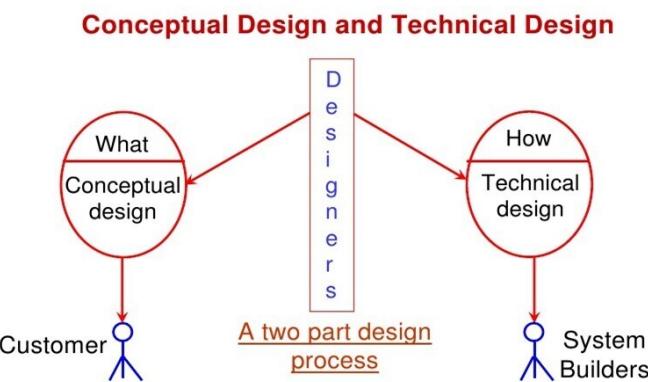
Если вы хотите построить дом, вы не забьёте и гвоздя без проекта.

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.

Software Design



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 представлены 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 несущественные 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 устраниет 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 основных характеристик для какой-либо концепции, имеет смысл определить все эти детали в классе, названном соответственно концепцией.

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 использовать 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 одинаковый 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, пользуясь ли я наследованием, 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?

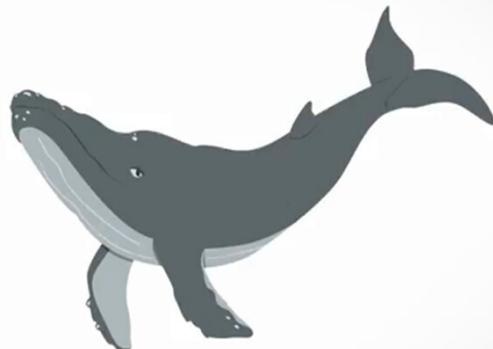
```
public class Animal {  
    private int numberofLegs;  
    private boolean hasTail;  
    /* Other characteristics of an animal can be included here */  
  
    public Animal(int legs, boolean tail) {  
        this.numberofLegs = legs;  
        this.hasTail = tail;  
    }  
  
    public void 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?

```
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 заменяют 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 ответственостей 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 поддерживающих 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

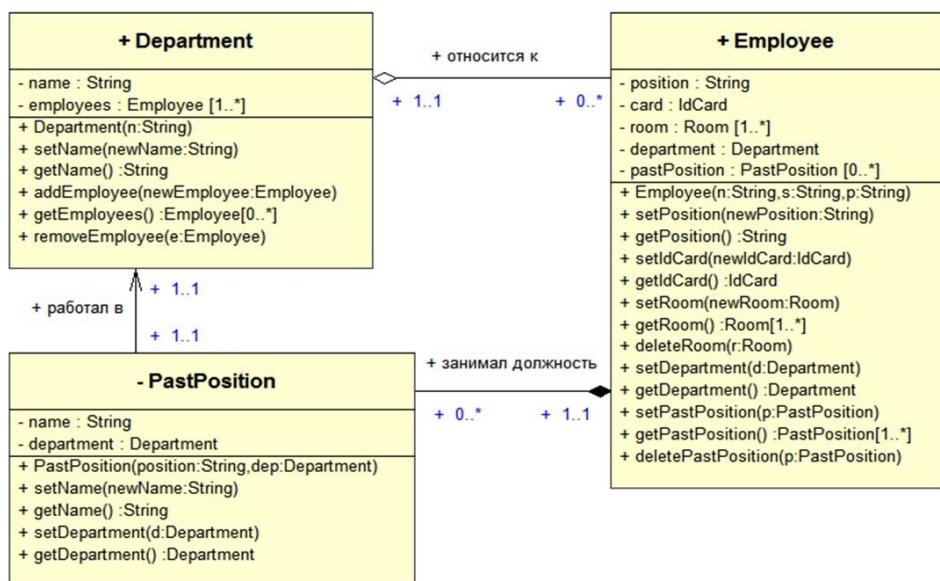
При проектировании здания архитекторы создают эскизы, чтобы визуализировать и экспериментировать с различными проектами.

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.

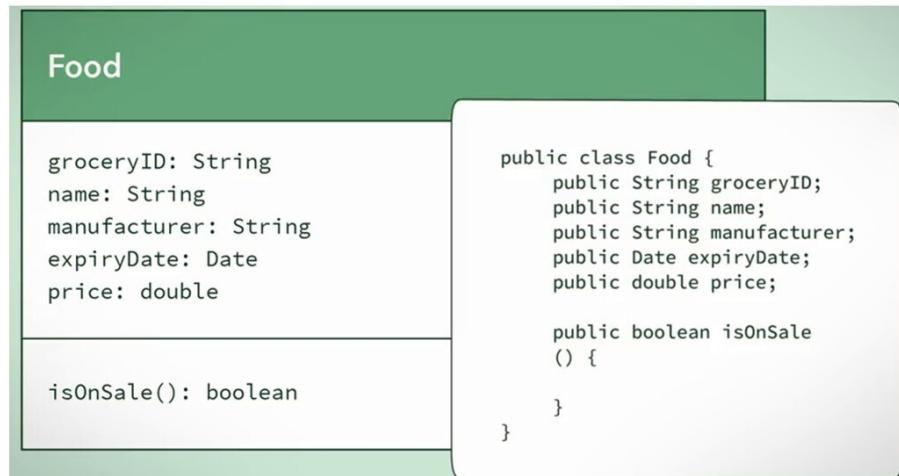
Некоторые из них стали операцией.

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 классов, 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.

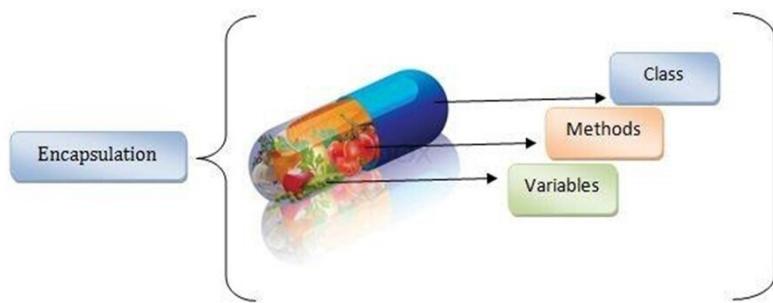
Детали будут отвлекать и отнимать много времени, при создании первоначальных проектов.

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.

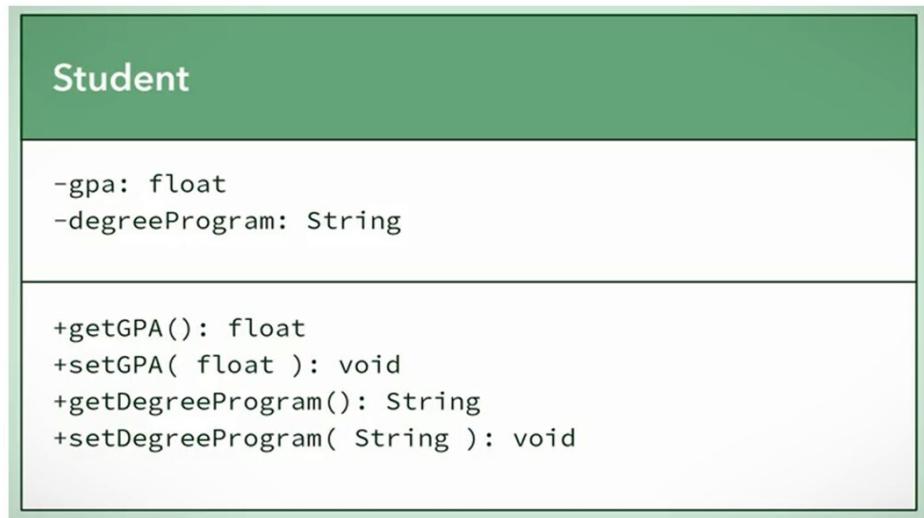
Во-вторых, вы можете предоставить определенные данные и функции этого объекта, чтобы к ним можно получить доступ из других объектов.

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.

Эти знаки минуса указывают, что метод или атрибут является приватным.

Доступ к приватным атрибутам возможен только из класса.

Вне этого класса вместо того, чтобы напрямую манипулировать атрибутами, вы должны устанавливать их значения через публичные методы.

Благодаря тому, что данные объекта могут управляться с помощью публичных методов, вы можете контролировать, как и когда эти данные будут доступны.

И вы разрешаете доступ только к предоставленным вами данным.

Эта процедура обеспечивает защиту от недобросовестных изменений.

Методы getter – это методы, которые извлекают данные, и их имена обычно начинаются с get и заканчиваются именем атрибута, значение которого вы будете возвращать.

Сеттер методы изменяют данные, и их имена обычно начинаются с set и заканчиваются именем переменной, которую вы хотите установить.

Сеттер методы используются для безопасного назначения приватного атрибута.

И эти методы обеспечивают целостность данных.

Чтобы изменить часть данных, вам необходимо использовать одобренный способ.

И доступ к данным должен осуществляться одобренным способом.

Единственный способ манипулировать скрытыми данными – это публичные функции, которые разрешают доступ к ним.

```
public class Student {  
    private float gpa;  
    private String degreeProgram;  
  
    public float getGPA() {  
        return gpa;  
    }  
  
    public void setGPA(float newGPA) {  
        gpa = newGPA;  
    }  
  
    public String getDegreeProgram() {  
        return degreeProgram;  
    }  
  
    public void setDegreeProgram( String newDegreeProgram ){  
        degreeProgram = newDegreeProgram;  
    }  
}
```

Соответственно код для диаграммы будет выглядеть следующим образом.

У вас может быть ситуация, при которой существуют ограничения на изменение программы студента.

И вы можете изменить свой код, чтобы более точно отобразить это ограничение, когда дело дойдет до изменения программы в методе.

И внешнему наблюдателю не нужно знать, как реализуются публичные методы.

Это означает, что вы можете добавить дополнительный код к своим Getter и Setter методам, если вам это нужно.

Инкапсуляция предназначена для защиты вашего класса и его объектов.

Она также позволяет использовать интерфейс одобренных методов для других классов для безопасного использования класса.

И инкапсуляция позволяет скрыть детали реализации от других классов.

Принцип Декомпозиции в UML

Теперь, давайте рассмотрим, как применять декомпозицию.

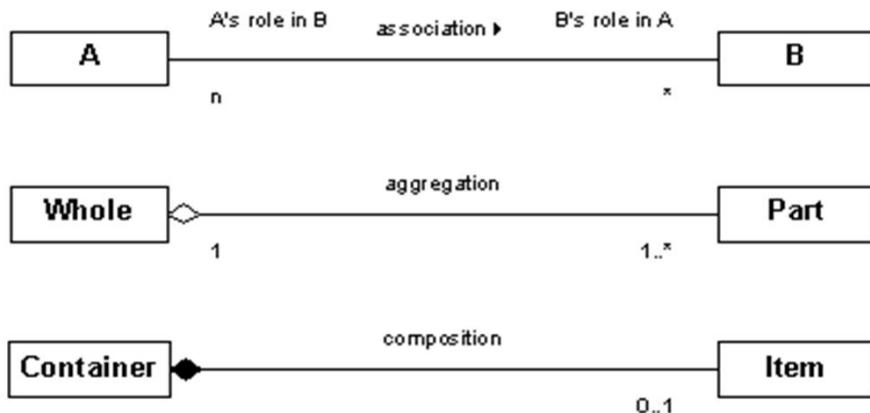
Декомпозиция берет целую вещь и делит ее на разные части.

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

Как это выглядит при программировании?

Есть ли способ расширить это определение и сделать его более конкретным?

В декомпозиции существуют три типа отношений – композиция, ассоциация, и агрегация.



Они определяют взаимодействие между целым и частями.

Давайте рассмотрим эти взаимодействия с нескольких разных точек зрения, в Java-коде, диаграмме классов UML и определении.

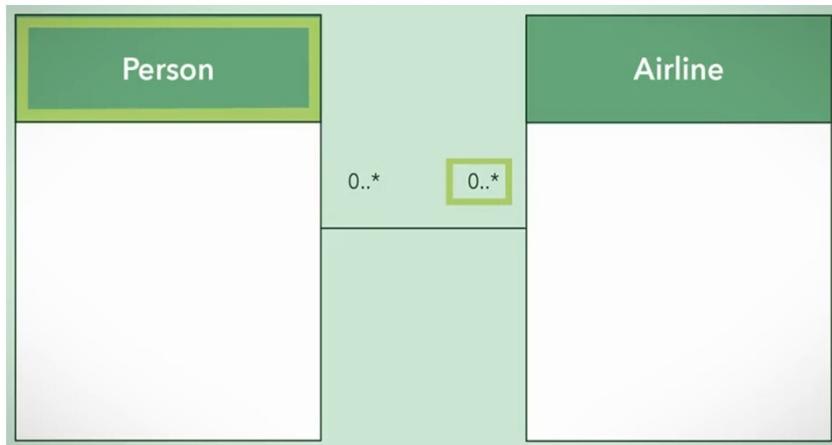
Первое отношение декомпозиции, которое мы рассмотрим, является ассоциацией.

Ассоциация – это некоторые отношения.

Это означает, что между двумя объектами существует свободная связь.

Эти объекты могут взаимодействовать друг с другом в течение некоторого времени.

Например, объект класса может использовать службы или методы, предоставляемые объектом другого класса.



Это похоже на отношения между человеком и авиакомпанией.

Человек обычно не владеет авиакомпанией, но может взаимодействовать с ней.

Авиакомпания также может взаимодействовать со многими объектами.

Есть несколько человек и несколько авиакомпаний, и они не зависят друг от друга.

Каждая UML-диаграмма считается объектом.

Прямая линия между двумя объектами UML означает, что взаимосвязь между ними является ассоциацией.

Нотация или обозначение $0 .. *$ по обе стороны от отношения означает, что объект человека связан с нолем или более объектов авиакомпании.

И объект авиакомпании связан с нолем или более объектами человека.

Каждое из отношений ассоциации находится между полностью отдельными объектами.

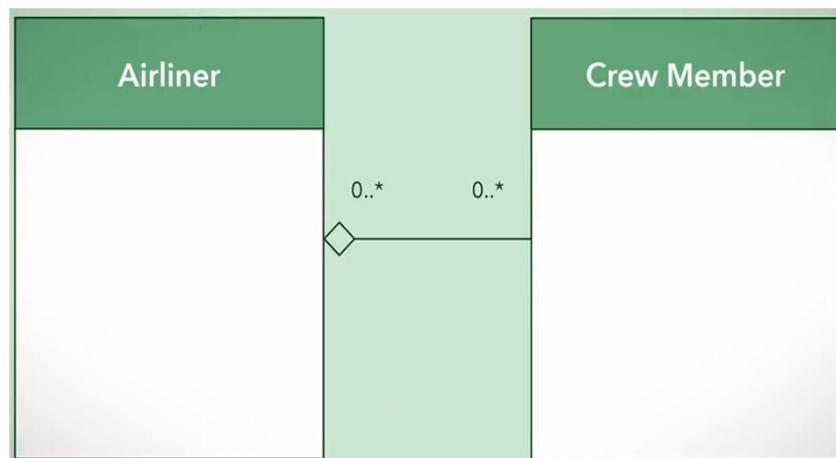
Если один объект будет уничтожен, другой может продолжать существовать, в отличие от человека и его органа.

В отношении ассоциации может быть любое количество каждого элемента.

Один объект не принадлежит другому.

```
public class Student{  
    public void play(Sport sport){  
        ...  
    }  
}
```

Теперь давайте рассмотрим пример ассоциации в коде.
В этом отрывке кода студенту передается объект спорта для игры.
Студент не обладает им, а играет в него.
Это отношения между двумя совершенно отдельными объектами.
Студент может играть в любое количество видов спорта.
И любое количество студентов может заниматься спортом.
В целом, ассоциация – это свободное партнерство между двумя объектами, которые существуют полностью независимо друг от друга.
Следующее отношение декомпозиции, – это агрегация.



Агрегация – это отношение has-a, в котором целое имеет части, принадлежащие ему.
В этих отношениях может быть общее использование отдельных частей разными целыми частями.

И отношение has-a от целого к частям считается слабым.

Это означает, что, хотя части могут принадлежать целым, они также могут существовать независимо.

Это похоже на отношения между авиалайнером и его экипажем.

Важной частью авиалайнера является его экипаж.

Без экипажа авиалайнер не сможет летать.

Однако авиалайнер не прекращает свое существование, если на борту нет экипажа.

То же самое касается экипажа, они являются частью работы лайнера, но экипаж не прекращает свое существование и не разрушается, если они не находятся на борту своего авиалайнера.

Эти сущности имеют отношения, но могут существовать вне друг друга.

Эта диаграмма UML описывает отношения между авиалайнером и экипажем.

Здесь говорится, что объект авиалайнера имеет ноль или более членов экипажа.

Кроме того, объект члена экипажа может быть доставлен на ноль или более объектов авиалайнера.

Пустой ромб обозначает, какой объект считается целым, а не частью отношения.

Этот пустой алмаз является символом агрегации.

Каждая агрегация имеет отношения «есть».

У авиалайнера есть экипаж.

Но эти отношения слабые.

Если один из объектов в отношении разрушен, другой может продолжать существовать.

```
public class Airliner {  
    private ArrayList<CrewMember> crew;  
  
    public Airliner(){  
        crew = new ArrayList<CrewMember>();  
    }  
  
    public void add( CrewMember crewMember ){ ... }  
}
```

Теперь посмотрим на пример кода для агрегации.

В классе авиалайнеров есть список членов экипажа. И список членов экипажа инициализируется пустым.

И публичный метод позволяет добавлять новых членов экипажа.

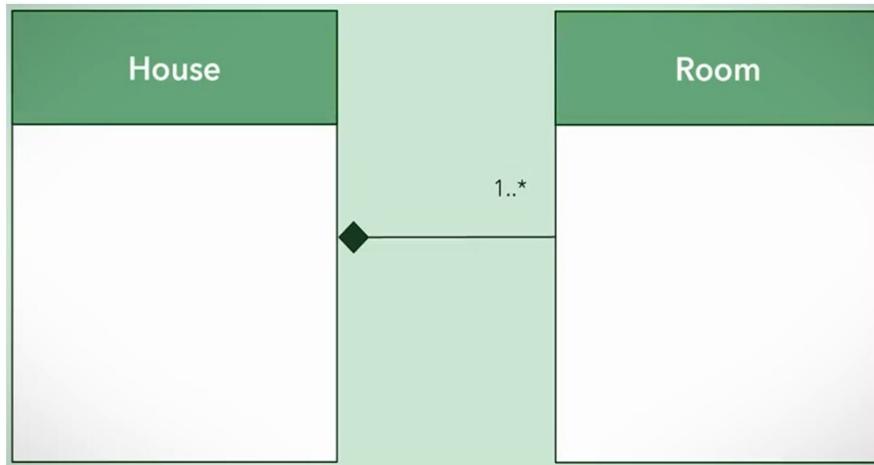
У авиалайнера есть экипаж. Это означает, что у авиалайнера может быть ноль или более членов экипажа.

Агрегация – это слабое has-a отношение между классами.

Один объект имеет другой, но объекты слабо связаны.

Они могут существовать один без другого.

Следующий тип отношений декомпозиции, это композиция.



Композиция – это эксклюзивное объединение частей, известное как сильное отношение has-a.

Это означает, что целое не может существовать без его частей.

Если теряется какая-либо часть целого, все перестает существовать.

Если целое уничтожено, то все его части также будут уничтожены.

Например, если взять отношение между домом и комнатой.

Дом состоит из нескольких комнат. Однако, если бы вы уберете дом, его комнаты также перестанут существовать.

У вас не может быть комнаты без дома.

Эта диаграмма UML описывает взаимосвязь между домом и комнатой, и показывает, что объект дома имеет один или несколько объектов комнаты.

Заполненный ромб рядом с домом означает, что дом является целым в этом отношении.

Если ромб заполнен, это означает, что отношения «сильные».

Два связанных объекта не могут существовать друг без друга.

И заполненный ромб означает, что это композиция.

```
public class Human{  
    private Brain brain;  
  
    public Human(){  
        brain = new Brain();  
    }  
}
```

Это пример композиции в коде Java.

В этом примере мозг создается в то же время, что и объект человека.

Мозг нельзя создать где-то еще, и он не должен передаваться в объект человека при создании в конструктор.

Мозг автоматически создается вместе с человеком.

Две части – человек и мозг тесно зависят друг от друга, и не могут существовать друг без друга.

Таким образом, декомпозиция – это просто объекты целого, содержащие объекты частей.

В зависимости от вашего дизайна вы можете связывать целое с частями сильнее или слабее.

Это может быть ассоциация, очень свободное взаимодействие между двумя полностью независимыми объектами.

Агрегация, когда целое имеет часть, но оба могут жить независимо друг от друга.

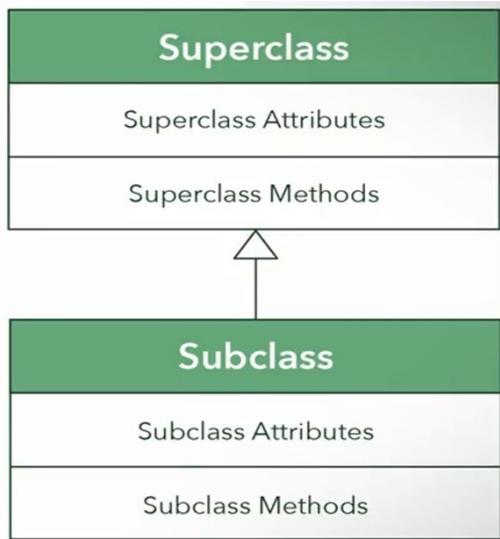
И, наконец, композиция, когда целое не может существовать без его частей и наоборот.

Принцип Обобщения в UML

Принцип обобщения может применяться в контексте наследования и в контексте интерфейсов.

Сначала давайте рассмотрим обобщение с помощью наследования.

Также, как и другие принципы проектирования, UML позволяет моделировать обобщение наследованием классов в системе.



Отображение наследования очень просто в диаграмме UML.

Вы просто соединяете два класса прямой стрелкой.

Это указывает на то, что два класса связаны наследованием.

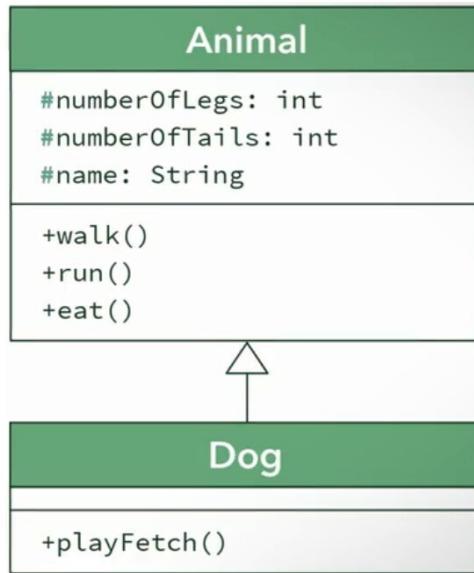
Суперкласс находится во главе стрелки, а подкласс – в хвосте.

Подумайте об этом, как о генеалогическом дереве, где младшее поколение находится внизу, а более старое поколение обычно расположено ближе к верхней части.

Стрелка используется для передачи наследования, что подразумевает, что подкласс будет иметь атрибуты и методы суперкласса.

Суперклассы – это обобщенные классы, а подклассы – это специализированные классы.

Теперь, когда вы знаете, как показать наследование в диаграмме классов UML, попробуем установить связь между UML и кодом.



Предположим, у нас есть класс собаки и класс животных.

И сначала мы можем смоделировать класс собак и класс животных на диаграмме UML, чтобы показать взаимосвязь между ними.

Мы также включим атрибуты и поведение обоих классов.

Это покажет, как два класса связаны друг с другом, как обобщается суперкласс и как специализируется подкласс.

Диаграмма классов UML описывает класс собаки как подкласс и класс животных как суперкласс.

Это означает, что класс собаки наследуется от класса животных.

Символ решетки используется для обозначения того, что атрибуты животных являются защищенными или `protected`.

В Java к защищенному атрибуту или методу может быть доступ только из самого инкапсулирующего класса, из всех его подклассов, и всех классов в одном пакете.

В Java пакет – это просто средство, с помощью которого классы могут быть организованы в пространство имен.

Мы знаем, что подкласс будет иметь все атрибуты и поведение суперкласса, которые он наследует.

Поэтому нам не нужно помещать атрибуты и поведение суперкласса в подкласс в нашей диаграмме UML.

Это связано с тем, что нотация наследования говорит нам, что подкласс уже имеет атрибуты и поведение, перечисленные в суперклассе.

```
public abstract class Animal {  
    protected int number_ofLegs;  
    protected int number_ofTails;  
    protected String name;  
  
    public Animal(String petName, int legs, int tails) {  
        this.name = petName;  
        this.number_ofLegs = legs;  
        this.number_ofTails = tails;  
    }  
  
    public void walk() { ... }  
    public void run() { ... }  
    public void eat() { ... }  
}
```

Теперь давайте преобразуем UML-модель в код классов животных и собак.

Так как животное является обобщением конкретных видов, мы не хотим, чтобы класс животных мог сам создать объект животного.

Поэтому мы используем ключевое слово `abstract`, чтобы объявить, что экземпляр этого класса не может быть создан.

И класс `Animal` будет суперклассом для подкласса собак.

И любой класс, который наследуется от класса `Animal`, будет иметь его атрибуты и поведение.

```
public class Dog extends Animal {  
    public Dog(string name, int legs, int tails) {  
        super(name, legs, tails);  
    }  
  
    public void playFetch() { ... }  
}
```

Это означает, что, если мы введем подкласс кошек в нашу систему, унаследованный от класса `Animal`, подклассы кошек и собак имели бы одинаковые атрибуты и поведение суперкласса животных.

Как и следовало ожидать, нам не нужно объявлять какие-либо атрибуты и поведения, которые класс собаки наследует от класса Animal.

Обратите внимание, что наша диаграмма UML и код аналогичны с точки зрения того, какие атрибуты и методы объявлены в суперклассе и подклассе.

Диаграмма UML представляет наш дизайн.

Мы объявляем наследование в Java с использованием ключевого слова «extends».

И вы создаете объекты из класса с помощью конструкторов.

С наследованием, если вы хотите получить экземпляр подкласса, вам нужно дать суперклассу возможность соответствующим образом подготовить атрибуты для объекта.

Классы могут иметь неявные конструкторы или явные конструкторы.

При использовании неявного конструктора или конструктора по умолчанию, всем атрибутам присваивается нуль.

Класс Animal в этой реализации имеет явный конструктор, который позволяет нам создать экземпляр животного с любым количеством ног.

Явные конструкторы используют, чтобы мы могли назначать значения атрибутам во время создания экземпляра.

Конструктор подкласса должен вызвать конструктор его суперкласса, если суперкласс имеет явный конструктор.

В противном случае атрибуты суперкласса не будут правильно инициализированы.

И чтобы получить доступ к атрибутам, методам и конструкторам суперкласса, подкласс использует ключевое слово super.

Подклассы могут переопределять методы своего суперкласса, и это означает, что подкласс может предоставить свою собственную реализацию для метода унаследованного суперкласса.

В Java суперкласс может иметь несколько подклассов.

Но подкласс может наследовать только один суперкласс.

Таким образом, мы можем создавать специализированные классы, такие как подклассы «собака» и «кошка», с индивидуальным или специальным поведением.

И обратите внимание, что сам подкласс может быть суперклассом для другого класса.

Наследование позволяет обобщить связанные классы в один суперкласс и допустить, чтобы подклассы сохраняли один и тот же набор атрибутов и поведения.

Это помогает устраниТЬ избыточность в коде и упростить реализацию изменений.

Класс обозначает тип для своих объектов.

Тип означает, что эти объекты могут делать с помощью своих публичных методов.

Например, экземпляры класса собаки – это объекты собаки, и эти объекты выполняют собачьи дела.

При моделировании задачи мы можем захотеть представить отношение подтипа между двумя типами.

Например, мы можем иметь тип собаки как подтип типа животного.

Это означает, что объект собаки не только является собакой, но также является и животным.

Таким образом, собачий объект ведет себя не только как собака, но он также должен вести себя как животное.

По сути, собака – это животное.

В JAVA наследование классов с ключевым словом extends используется для подтипования.

Если подкласс собаки расширяет суперкласс животного, объект собаки ведет себя не только как собака, он также будет вести себя как животное по умолчанию, через унаследованные методы и атрибуты животного.

Интерфейс JAVA также обозначает тип.

```
public interface IAnimal {  
    public void move();  
    public void speak();  
    public void eat();  
}  
  
public class Dog implements IAnimal {  
    /* Attributes of a dog can go here */  
  
    public void move() { ... }  
  
    public void speak() { ... }  
  
    public void eat() { ... }  
}
```

Однако, в отличие от класса, интерфейс только объявляет сигнатуры методов, а не конструкторы, атрибуты или тела методов.

Он определяет ожидаемое поведение в сигнатурах метода, но не предоставляет каких-либо деталей реализации.

В JAVA интерфейс также используется для подтиповирования.

Если класс собаки реализует интерфейс животных, то объект собаки ведет себя не только как собака, но также ожидается, что он будет вести себя как животное, предоставив все тела методов для сигнатур метода, перечисленных в интерфейсе.

Как и в случае с наследованием, собака – это животное.

Однако разница заключается в том, что класс собаки должен предоставить детали реализации для того, что значит быть животным.

Таким образом, интерфейс подобен контракту, который должен выполняться посредством реализации классов.

Как в случае наследовании, так и в случае интерфейса вы достигаете согласованности между типом собаки и животным, чтобы объект собаки использовался в любом месте вашей программы, когда вы имеете дело с типом животного.

В отличие от наследования, интерфейсы не являются обобщением набора классов.

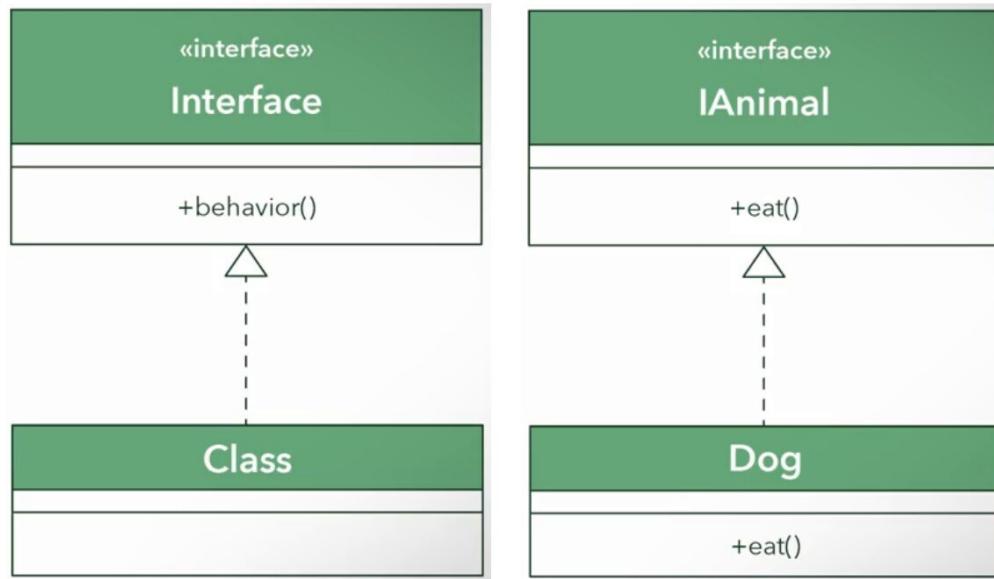
Важно понимать, что интерфейсы не являются классами.

Они используются для описания поведения.

Все, что содержит интерфейс, это сигнатуры методов.

Еще одна вещь, которую вы могли заметить, заключается в том, что интерфейс не инкапсулирует ни один из атрибутов животного.

Это связано с тем, что атрибуты не являются поведением.



В UML интерфейсы представляются аналогично классам.

Интерфейсы явно обозначаются в диаграммах UML с использованием двойных скобок.

Взаимодействие между интерфейсом и классом, реализующим интерфейс, указывается пунктирной стрелкой.

Класс находится в хвосте стрелки, а интерфейс находится в голове стрелки.

Это означает, что класс реализует интерфейс.

Как и абстрактные классы, которые являются классами, экземпляры которых не могут быть созданы, интерфейсы – это средство, с помощью которого вы можете реализовать полиморфизм.

В объектно-ориентированных языках полиморфизм – это когда два класса имеют одно и то же описание поведения, но реализации поведения могут быть разными.

И интерфейсы могут наследовать от других интерфейсов.

И точно так же, как с наследованием класса, наследованием интерфейса не следует злоупотреблять.

Это означает, что вы не должны расширять интерфейсы, если просто пытаетесь создать более крупный интерфейс.

Интерфейс А должен наследовать от интерфейса В, если поведение в интерфейсе А можно полностью использовать в качестве замены для интерфейса В.

Интерфейс также позволяет реализовать еще одну форму наследования, а именно множественное наследование.

Это когда подкласс может наследовать от одного или нескольких интерфейсов.

JAVA не поддерживает множественное наследование классов.

Это связано с тем, что наследование от двух или более суперклассов может привести к неоднозначности данных.

Когда ваш подкласс наследует от двух или более суперклассов, которые имеют атрибуты с одним и тем же именем или поведением с одной и той же сигнатурой метода, как вы различите их?

Так как JAVA не может определить, на какой из них вы ссылаетесь, множественное наследование не допускается.

Интерфейсы не создают эту проблему.

В JAVA класс может реализовать столько интерфейсов, сколько захочет.

Это связано с природой интерфейсов.

Так как они являются только контрактами и не обеспечивают конкретного способа для реализации этих контрактов, перекрывающиеся сигнатуры методов не являются проблемой.

Может существовать единая реализация для нескольких интерфейсов с перекрывающимися контрактами.

Интерфейсы позволяют описывать поведение без необходимости его реализации, что позволяет повторно использовать абстракции.

Вопросы

Вопрос 1

Некоторые из ранних языков программирования поддерживали:

Локальные переменные

Основную программу и подпрограммы +

Абстрактные типы данных

Объекты и классы

Вопрос 2

Каковы преимущества объектно-ориентированного программирования на языке Java?

Абстрактные типы данных +

Вычислительная эффективность

Имитирует реальную структуру проблемы +

Управление данными +

Вопрос 3

Сэму было предложено создать класс DeliveryDriver.

Сэм думал о проблеме и сводил ее к самым важным аспектам, таким как takeOrder, DeliveryArea и т. д. Он игнорировал вещи, которые не были важны в контексте, например, высота или цвет водителя. Он просто применил важный объектно-ориентированный принцип дизайна. Какая из этих концепций лучше всего описывает то, что он только что сделал?

DeliveryDriver

deliveryArea : DeliveryArea
~~eyeColour : Color~~
~~height : int~~

takeOrder()

Декомпозиция

Инкапсуляция

Обобщение

Абстракция +

Вопрос 4

Сэм определил важные атрибуты и поведение водителя доставки и поместил их в класс DeliveryDriver, например, «takeOrder», «deliverOrder» и «DeliveryArea». Он выставил некоторые из них другим классам. Он просто продемонстрировал важный принцип объектно-ориентированного дизайна. Какая из этих концепций лучше всего описывает то, что он сделал?

Инкапсуляция +

Декомпозиция

Абстракция

Обобщение

Вопрос 5

Сэм решил, что класс DeliveryDriver становится слишком сложным, поэтому он разделил его, переместив его поведение в несколько связанных классов, таких как DeliveryCar и DeliveryOrder., какой важный объектно-ориентированный принцип дизайна он продемонстрировал?

Обобщение

Инкапсуляция

Абстракция

Декомпозиция +

Вопрос 6

Сэм понял, что в классе DeliveryDriver есть поведение и атрибуты, которые могут использоваться другими классами, такими как BusDriver и TaxiDriver, поэтому он сделал класс под названием Driver, из которого эти классы унаследовали поведение. Какой объектно-ориентированный принцип дизайна он использовал?

Инкапсуляция

Декомпозиция

Абстракция

Обобщение +

Вопрос 7

Какие ключевые слова позволяют вашим классам достичь полиморфизма в Java?

Extends +

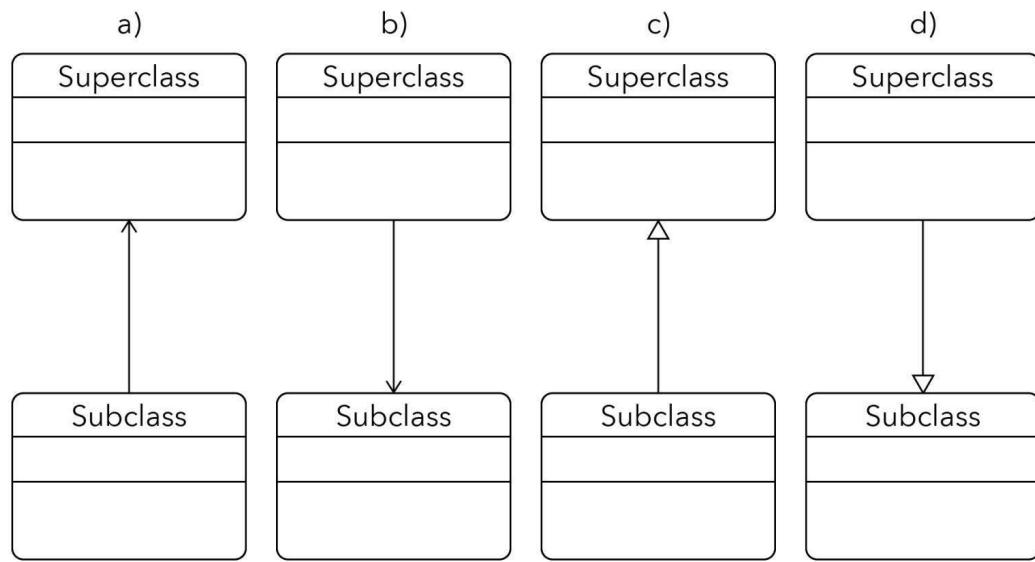
interfaces

implements +

inherits

Вопрос 8

Каков правильный способ показать наследование в диаграмме классов UML?



- a)
- b)
- c) +
- d)

Вопрос 9

Каков правильный способ показать абстрактный метод в диаграмме классов UML?

anOperation() +

«anOperation()»

#anOperation()

abstract anOperation()

Вопрос 10

Какая диаграмма класса UML является хорошим примером инкапсуляции?

a)	<table border="1"><tr><td>Coffee</td></tr><tr><td>+temperature: int +strength: int</td></tr><tr><td>+getTemperature(): int +setTemperature(int) +getStrength(): int +setStrength(int)</td></tr></table>	Coffee	+temperature: int +strength: int	+getTemperature(): int +setTemperature(int) +getStrength(): int +setStrength(int)	b)	<table border="1"><tr><td>Coffee</td></tr><tr><td>-temperature: int -strength: int</td></tr><tr><td>+getTemperature(): int +setTemperature(int) +getStrength(): int +setStrength(int)</td></tr></table>	Coffee	-temperature: int -strength: int	+getTemperature(): int +setTemperature(int) +getStrength(): int +setStrength(int)
Coffee									
+temperature: int +strength: int									
+getTemperature(): int +setTemperature(int) +getStrength(): int +setStrength(int)									
Coffee									
-temperature: int -strength: int									
+getTemperature(): int +setTemperature(int) +getStrength(): int +setStrength(int)									
c)	<table border="1"><tr><td>Coffee</td></tr><tr><td>-temperature: int -strength: int</td></tr><tr><td>+getTemperature(): int +setTemperature(int) +getStrength(): int +setStrength(int)</td></tr></table>	Coffee	-temperature: int -strength: int	+getTemperature(): int +setTemperature(int) +getStrength(): int +setStrength(int)	d)	<table border="1"><tr><td>Coffee</td></tr><tr><td>-temperature: int -strength: int</td></tr><tr><td>-getTemperature(): int -setTemperature(int) -getStrength(): int -setStrength(int)</td></tr></table>	Coffee	-temperature: int -strength: int	-getTemperature(): int -setTemperature(int) -getStrength(): int -setStrength(int)
Coffee									
-temperature: int -strength: int									
+getTemperature(): int +setTemperature(int) +getStrength(): int +setStrength(int)									
Coffee									
-temperature: int -strength: int									
-getTemperature(): int -setTemperature(int) -getStrength(): int -setStrength(int)									

- a)
- b) +**
- c)
- d)

Вопрос 11

Добавьте необходимое ключевое слово, чтобы заполнить это объявление переменной экземпляра в классе, если вы используете принцип инкапсуляции, чтобы скрыть эту переменную от всех других классов:

_____ String message;

Вопрос 12

Существуют три разных типа отношений, которые могут иметь объекты. Какой из них лучше всего описывает отношения между стулом и его ножками?

Ассоциация

Агрегация

Композиция +

Формирование

Связанность и когезия

Мы уже обсудили четыре принципа объектно-ориентированного моделирования и программирования – абстракцию, инкапсуляцию, декомпозицию и обобщение.

Мы также увидели, как представить объектно-ориентированную модель, используя UML диаграмму классов и код Java.

Диаграммы классов отлично подходят для того, чтобы отобразить структуру задачи и технический дизайн программного решения.

Теперь мы поговорим об оценке сложности дизайна.

Средний человек может хранить только семь вещей в краткосрочной памяти.

Это было установлено в психологическом исследовании Джорджа Миллера, в котором испытуемым приходилось вспоминать от 1 до 14 случайных звуков и изображений.

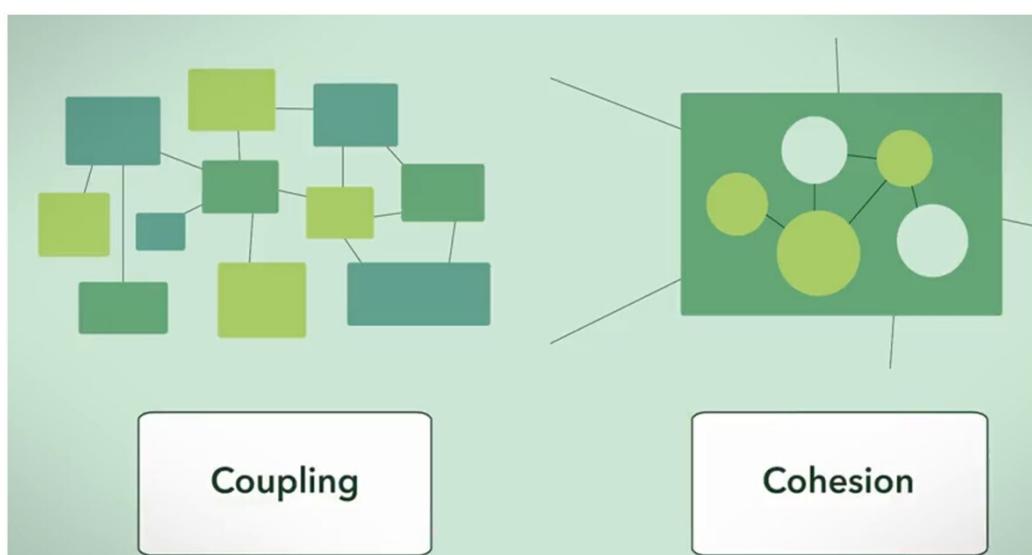
Испытуемые начинали испытывать трудности, когда число воспоминаний достигало 7.

Когда вы программируете, сохранение модулей простыми является критическим.

Как только вы сконструируете сложность, превышающую ту, которую разработчики могут мысленно обрабатывать, ошибки будут возникать чаще.

Поэтому у вас должен быть способ оценить сложность дизайна.

Понятия, которые мы будем использовать для оценки сложности проектирования, – это связанность и когезия или сцепление.



Связанность фокусируется на сложности между модулем и другими модулями.

Когезия или сцепление фокусируется на сложности в модуле.

Эти две идеи помогают лучше применять объектно-ориентированные принципы проектирования и создать более управляемую систему.

Когда вы разрабатываете систему, вы объединяете различные модули.

Подумайте о плохом дизайне, как о кусочках головоломки, где ваши модули – это кусочки.

Вы только можете связать кусочек головоломки только с другим конкретным кусочком головоломки и больше ничего.

С другой стороны, давайте подумаем о хорошо продуманной системе, такой как блоки Lego.

Вы можете подключить любые два блока Lego без особых проблем, и все блоки Lego совместимы друг с другом.

При разработке системы вы захотите сделать это как Lego.

Таким образом, вы можете легко подключать и повторно использовать модули вместе.

Степень связанности для модуля отражает степень сложности подключения модуля к другим модулям.

Если ваш модуль сильно зависит от других модулей, вы можете сказать, что этот модуль тесно связан с другими модулями.

И это похоже на кусочки головоломки.

С другой стороны, если ваш модуль легко подключается к любым другим модулям, этот модуль слабо связан с другими модулями.

И это похоже на Лего.

Вы хотите, чтобы связывание для вашего модуля было свободным или слабым, но не сильным.

При оценке связанности модуля вам необходимо учитывать степень, легкость и гибкость.

Степень – это количество соединений между данным модулем и другими модулями.

При связывании вы хотите сохранить эту степень небольшой.

Например, если модуль подключается к другим модулям через небольшое количество параметров или узких интерфейсов, то степень будет небольшой, и связь будет свободной.

Легкость заключается в том, насколько простым является связывание данного модуля с другими модулями.

При связывании вы хотите, чтобы соединения были легко сделаны без необходимости понимания реализации других модулей.

Гибкость – это то, как взаимозаменямы другие модули для данного модуля.

При связывании вы хотите, чтобы другие модули могли быть легко заменены на лучшие в будущем.

Опять же, как и в Лего.

Связанность касается сложности между модулем и другими модулями, но вам также необходимо учитывать сложность в самом модуле.

Вот где вы рассматриваете когезию.

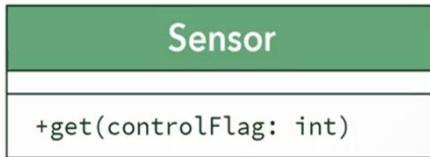
Когезия представляет собой ясность обязанностей модуля.

Если ваш модуль выполняет одну задачу и ничего другого или имеет четкую цель, ваш модуль обладает высокой когезией.

С другой стороны, если ваш модуль пытается инкапсулировать несколько целей или имеет нечеткую цель, ваш модуль имеет низкую когезию.

И вам нужна когезия.

Если вы обнаружите, что ваш модуль имеет более чем одну обязанность, вероятно, пора разделить ваш модуль.



```
public void get (int controlFlag) {  
    switch (controlFlag) {  
        case 0:  
            return this.humidity;  
            break;  
        case 1:  
            return this.temperature;  
            break;  
        default:  
            throw new UnknownControlFlagException();  
    }  
}
```

Давайте посмотрим на пример связывания и когезии.

Предположим, что у нас есть класс, называемый датчиком, который имеет две задачи: получение показаний влажности и получение показаний температуры.

Здесь у нас есть метод `get`, который принимает флаг нуль, если вы хотите вернуть значение влажности, и принимает флаг единицу, если вы хотите, чтобы метод возвращал значение температуры.

Теперь давайте оценим этот класс датчика на основе показателей связывания и когезии.

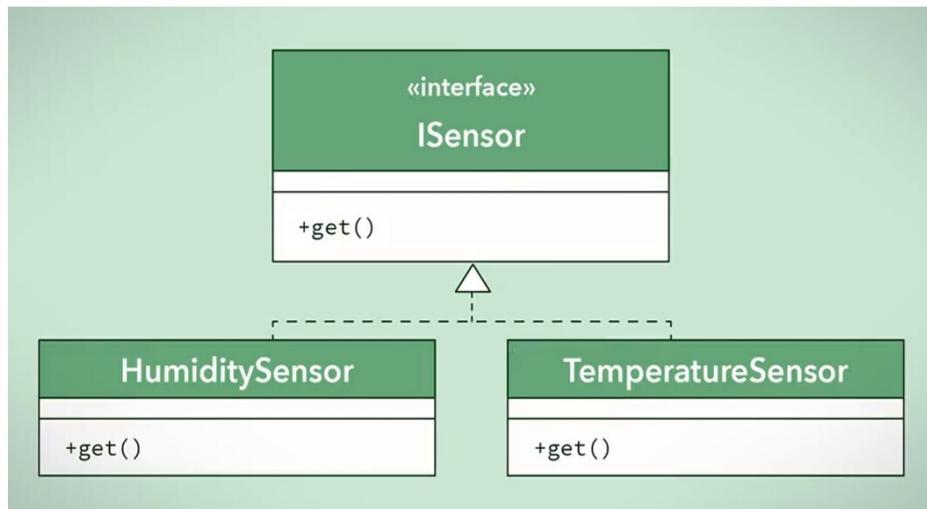
Так как класс датчика не имеет четкой цели, он имеет низкую когезию.

Посмотрите на этот метод `get`.

Легко ли определить, что он делает?

Так как неясно, что означает флаг управления, нам нужно прочитать код внутри самого метода, чтобы узнать, какие значения ему дать.

Таким образом, у этого метода отсутствует легкость, и мы должны нарушить инкапсуляцию, чтобы использовать этот метод.



Давайте посмотрим на новый дизайн той же системы.

Теперь класс датчика заменен классом датчика влажности и классом температурного датчика.

Каждый из этих классов имеет одну четко определенную цель.

Поскольку у каждого класса есть четкая цель, вы можете сказать, что эти классы имеют высокую когезию.

Метод `get` теперь не скрывает никакой информации, как раньше.

И нам не нужно нарушать инкапсуляцию, чтобы заглянуть внутрь метода.

Вы можете разумно предположить, что у датчика влажности метод возвращает влажность и у температурного датчика метод возвращает температуру.

Однако мы увеличиваем количество соединений между модулями.

Таким образом, мы должны обеспечить баланс между низким связыванием и высокой степенью когезии в дизайне.

Для сложной системы сложность может быть распределена между модулями или внутри модулей.

Так как модули упрощаются для достижения высокой когезии, им может потребоваться больше зависеть от других модулей, увеличивая связывание.

И так как связанность между модулями упрощается для достижения низкого связывания, возможно, им может потребоваться больше обязанностей, снижая когезию.

Разделение ответственостей

Целью применения принципов дизайна – помочь создать гибкую, многоразовую и поддерживаемую систему.

И одним из принципов дизайна является принцип разделения ответственостей.



Что такое ответственность?

Ответственность представляет собой очень общее понятие, в основном это нечто, что важно для решения задачи.

Давайте возьмем пример супермаркета.

Ответственностью супермаркета может быть предоставить продукты и принимать оплату.

Эти ответственности имеют значение при управлении бизнесом для обслуживания своих клиентов.

Но как организован супермаркет для работы с этими ответственостями?

В нем существуют отдельные отделы, каждый из которых сосредоточен на своей ответственности.

И каждая ответственность влечет за собой набор подзадач.

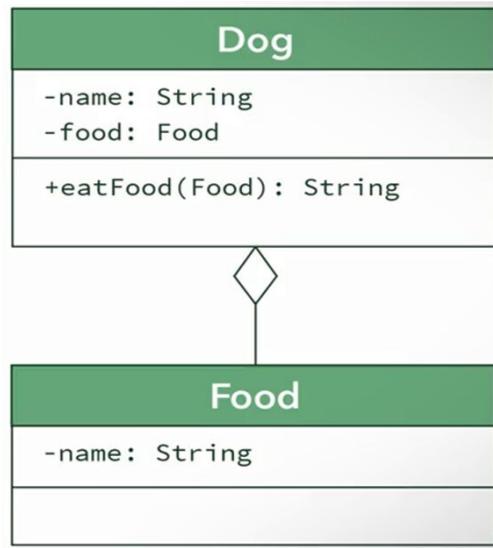
И каждый отдел знает, что делать и как работать со своей конкретной ответственностью.

Таким образом, организация супермаркета предусматривает разделение ответственостей.

Программная система решает задачу аналогичным образом.

Задача может быть сложной с большим количеством ответственостей.

И существуют ответственности, которые можно абстрагировать от области задачи.



Давайте посмотрим на поведение собаки.

Некоторые основные поведения, которые собака может выполнять, это ходить, бегать, гавкать и есть.

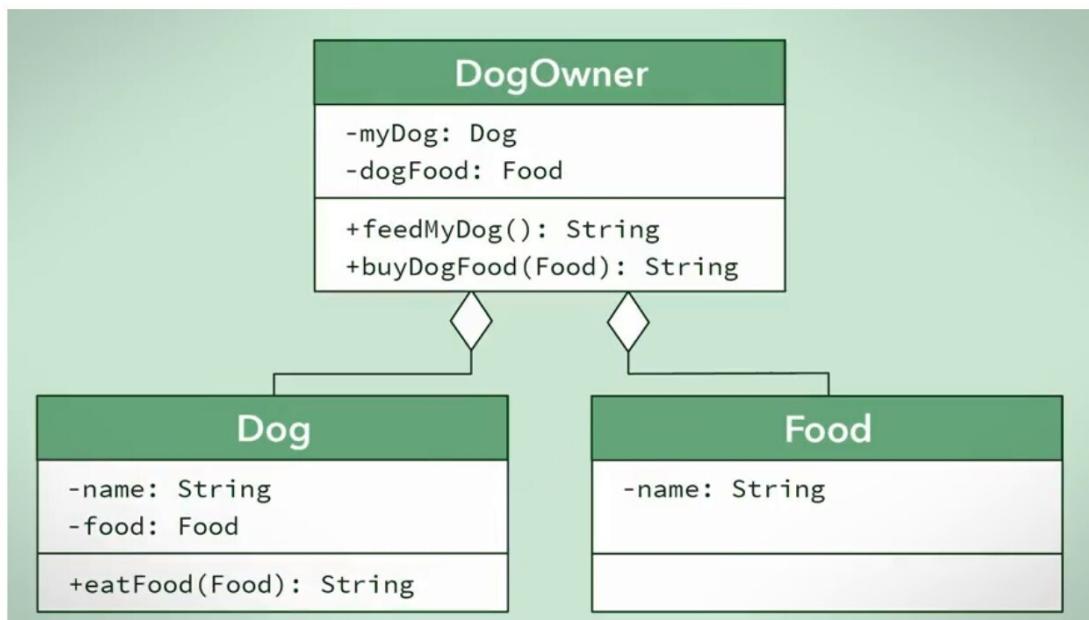
Хотя такое поведение легко идентифицировать и абстрагировать, нам нужно спросить себя, какое поведение собака может делать сама по себе?

И какое поведение нуждается в помощи от чего-то или кого-то другого?

Наша UML диаграмма говорит нам, что у собаки есть еда, которую она знает, как есть.

Мы можем сказать собаке есть пищу, давая ей пищу, но является ли это правильным способом моделирования ситуации?

Кто на самом деле дает собаке пищу? Всегда ли у собаки есть пища, или собака ест пищу, которую дает ей владелец?



На самом деле собаке понадобится владелец, чтобы ее накормить.

Собака знает, как есть пищу, но она ничего не знает о еде, которую она ест, пока ее владелец ее не кормит.

Нам нужно разделить две ответственности – питания и обеспечение питания.

Это можно сделать, представив класс владельца собаки.

В нашем новом дизайне класс собаки знает, как есть пищу.

А класс владельца собаки – знает, как получить корм для собак и как отдать его собаке.

Мы устранили ответственность того, как добыть пищу, из класса собаки и позволили владельцу собаки справиться с этой ответственностью.

При разделении ответственостей, мы должны инкапсулировать поведение и атрибуты в классах, которые связаны с данным поведением и атрибутами.

Это помогает нам создать модульную систему, в которой отдельные классы легко могут быть изменены без необходимости переписывать большую часть нашего кода.

Таким образом, наша цель – создать гибкий многоразовый и поддерживаемый код.

Разделение ответственостей создает более когезионные классы с использованием абстракции, инкапсуляции, декомпозиции и обобщения.

Это создает систему, которую легче поддерживать, потому что каждый класс организован так, чтобы он содержал только код, необходимый для выполнения своей работы.

В свою очередь увеличивается и модульность, что позволяет разработчикам повторно использовать и изменять отдельные классы, не затрагивая другие классы.

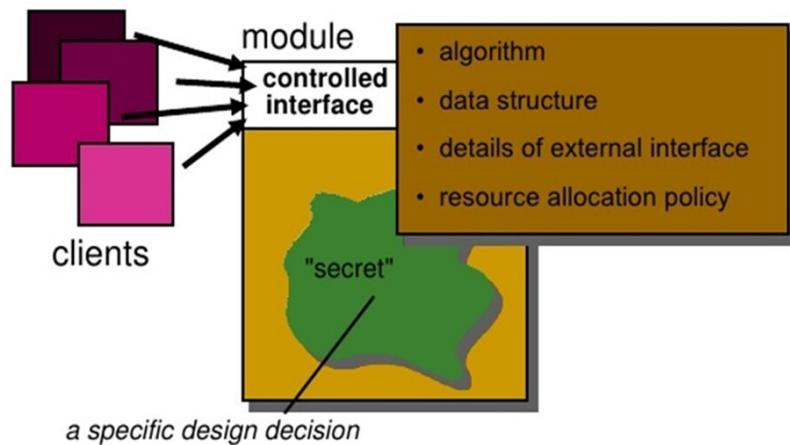
В нашем примере ясно, где находятся границы каждого класса.

Однако реальные задачи могут быть не такими очевидными.

И решение о том, как абстрагировать, инкапсулировать, декомпозировать и обобщать для выполнения ответственностей для данной задачи, лежит в основе разработки модульного программного обеспечения.

Скрытие информации

Еще один фактор, который нужно учитывать при проектировании системы, это доступ к информации.



Нужно иметь доступ только к той информации, которая необходима для выполнения работы.

Итак, как мы ограничиваем информацию для доступа различным модулям нашей системы?

Мы делаем это, применяя принцип сокрытия информации.

Информационное сокрытие позволяет модулям нашей системы давать другим модулям минимальный объем информации, необходимой для правильного их использования, и скрывать все остальное.

И для работы с модулем нет необходимости знать детали реализации этого модуля.

Можно использовать этот модуль только через интерфейс.

В общем и целом, вещи, которые могут измениться, например, детали реализации, должны быть скрыты.

И вещи, которые не должны меняться, раскрываются через интерфейсы.

Предположим, мы работаем в одной и той же системе, но в разных модулях.

Если один модуль требует информации из другого модуля, сокрытие информации позволяет предоставить другому только ту информацию, которая необходима для работы другого модуля.

Не нужно давать доступ ко всему в модуле, и не нужно знать, как работает этот модуль.

Информационное сокрытие часто связано с инкапсуляцией.

Мы используем инкапсуляцию для связывания атрибутов и поведения с их соответствующим классом и предоставления интерфейса для обеспечения доступа.

Инкапсуляция эффективно скрывает реализацию поведения, так как единственный доступ осуществляется через интерфейс определенных методов.

Другие классы могут полагаться только на информацию в сигнатурах этих методов, а не на их реализации.

Скрытие информации за счет инкапсуляции, позволяет нам изменять реализацию, не изменяя ожидаемый результат.

```
String a = "Hello";
String b = a.concat(" World");
```

Посмотрите на этот пример, в библиотеке String есть метод concat.

Но как он реализован?

Используются ли дополнительные структуры данных?

Или другие вызовы методов, к которым мы не имеем доступа?

Будучи пользователем библиотеки, вы получаете доступ к функциональности через интерфейс, а именно через конкретную сигнатуру метода concat.

Но вам не показано, как реализована эта функциональность.

Это связано с тем, что реализация может измениться, и вы не должны зависеть от того, как она работает.

И атрибуты или поля класса также могут быть скрыты, чтобы предотвратить изменение важной информации в классе.

Например, если атрибут имеет решающее значение для поведения класса, то мы не хотим, чтобы внешние классы меняли его напрямую.

Инкапсуляция – это практический принцип дизайна, используемый для реализации концепции скрытия информации.

И вы можете скрыть информацию с помощью модификаторов доступа.

Модификаторы доступа определяют, какие классы имеют доступ к атрибутам и поведению.

Они также определяют, какой атрибут и поведение суперкласса будет использоваться его подклассами.

С помощью информационного сокрытия вы контролируете, какой информацией вы хотите поделиться, и какое поведение вы хотите позволить видеть другим.

Вы раскрываете неизменяющиеся вещи через интерфейсы, и скрываете изменяющиеся вещи, такие как детали реализации.

Концептуальная целостность

Концептуальная целостность – это создание совместимого программного обеспечения.



Концептуальная целостность обеспечивает решения о том, как ваша система будет разработана и внедрена.

Так что даже если несколько человек работают над программным обеспечением, со стороны кажется, что только один ум руководит всей работой.

Теперь важно понимать, что концептуальная целостность не означает, что разработчики в команде не могут высказывать свое мнение о программном обеспечении.

Это больше касается соглашения использовать определенные принципы дизайна.

Существует несколько способов достижения концептуальной целостности.

Одним из важных способов является общение.

Это использование некоторых практик гибкой разработки, таких как ежедневные встречи и ретроспективы спринтов, где члены команды могут прийти к соглашению использовать определенные библиотеки или методы при решении определенных задач.

И где можно поддерживать согласованность кода.

Например, члены команды могут следовать определенному соглашению об именах.

Помимо общения, еще один способ обеспечить концептуальную целостность – это обзоры кода.

Обзоры кода – это систематические проверки написанного кода.

Обзоры кода часто используются для обнаружения ошибок в программном обеспечении, а также для того, чтобы разные разработчики соглашались свой код между собой.

При этом, разработчики оценивают «код» друг друга, чтобы выявить проблемы.

Кроме того, использование определенных принципов дизайна и конструкций программирования также может помочь в обеспечении концептуальной целостности.

Например, использование шаблонов проектирования.

Они предоставляют признанные структуры для ваших классов для решения задач дизайна и обеспечения согласованности кода.

Другой подход к достижению концептуальной целостности – это четко определенный дизайн или архитектура, лежащие в основе программного обеспечения.

В то время как разработка программного обеспечения обычно связана с внутренним проектированием программного обеспечения, работающего как один процесс, архитектура программного обеспечения описывает, как программное обеспечение, работающее как несколько процессов, работает вместе и как они соотносятся друг с другом.

Объединение концепций также является еще одним подходом к поддержанию концептуальной целостности.

И наконец, наличие небольшой основной группы, которая принимает изменения в коде, является еще одним подходом к достижению концептуальной целостности.

Это похоже на обзор кода, но ограничивает обзор только основными членами вашей команды разработчиков программного обеспечения.

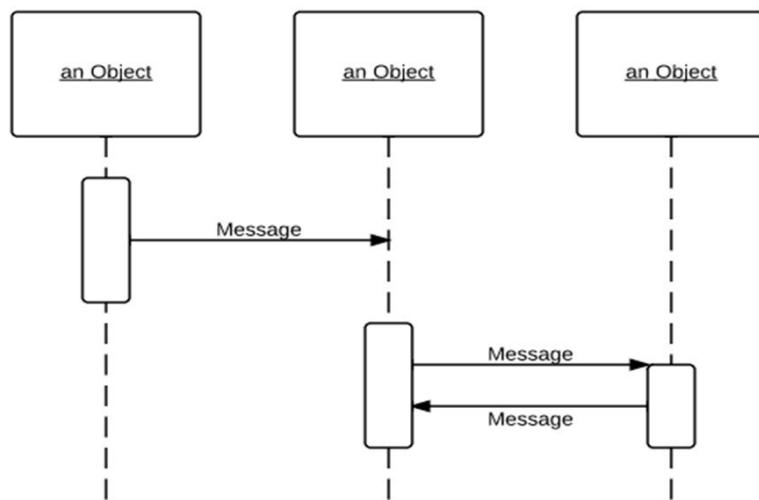
Эти участники несут ответственность за то, чтобы изменения программного обеспечения соответствовали общей архитектуре и дизайну программного обеспечения.

Концептуальная целостность часто упоминается как наиболее важное соображение при проектировании системы.

Фред Брукс, известный компьютерный архитектор, утверждает в своей книге «Мифический человек-месяц», что лучше иметь систему, которая упускает некоторые функции и улучшения, но отражает один набор идей дизайна, чем иметь систему, которая содержит много хороших, но независимых и несогласованных идей.

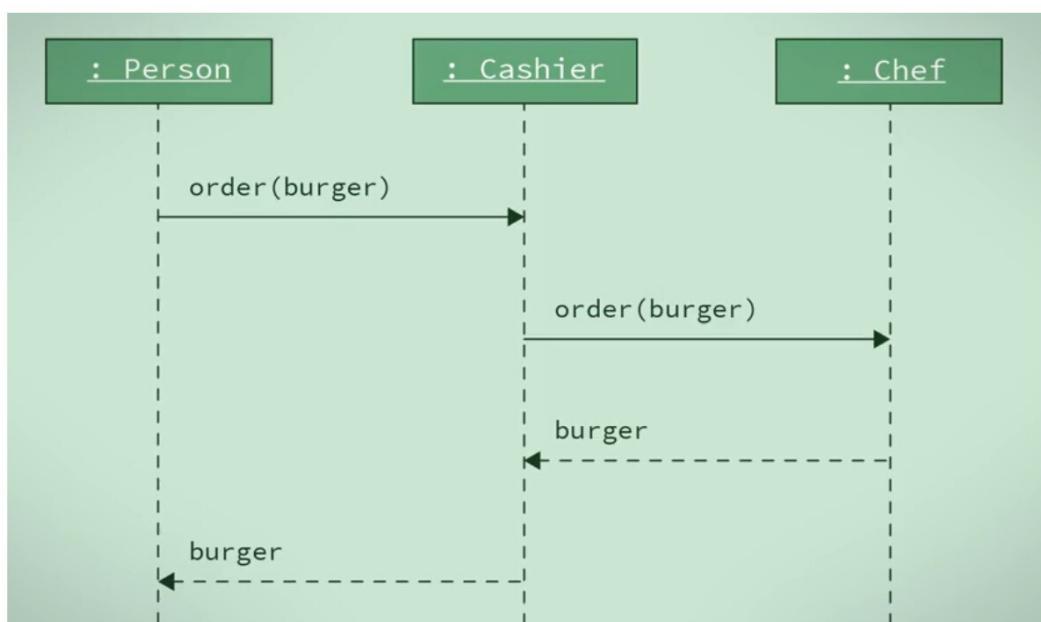
Моделирование поведения. UML диаграммы последовательности

Теперь обсудим диаграммы последовательности.



Диаграммы последовательности используются, чтобы показать, как объекты в программе взаимодействуют друг с другом для выполнения задач.

Проще говоря, думайте о диаграмме последовательности, как о карте разговоров между разными людьми, где эта карта отображает все сообщения, отправленные от человека к человеку.



Предположим, что человек хочет заказать гамбургер в местном ресторане быстрого питания.

И вот простая диаграмма последовательности.

Этот человек отправится в ресторан и поговорит с кассиром и закажет гамбургер.

Затем кассир поговорит с шеф-поваром, чтобы сообщить ему заказ.

И он приготовит гамбургер, и отдаст его кассиру.

А кассир передаст его человеку.

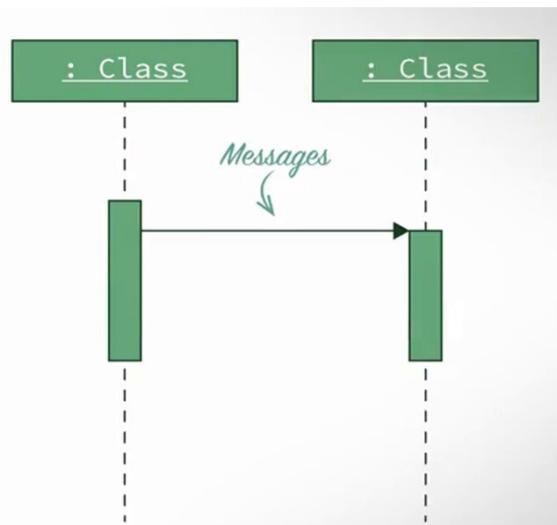
Знание того, как разбивать систему на классы, имеет важное значение для создания диаграмм последовательности.

Диаграмма последовательности описывает, как объекты в системе взаимодействуют для выполнения конкретной задачи.

При создании диаграмм последовательностей сначала вы используете поле для указания роли, которую играет объект.

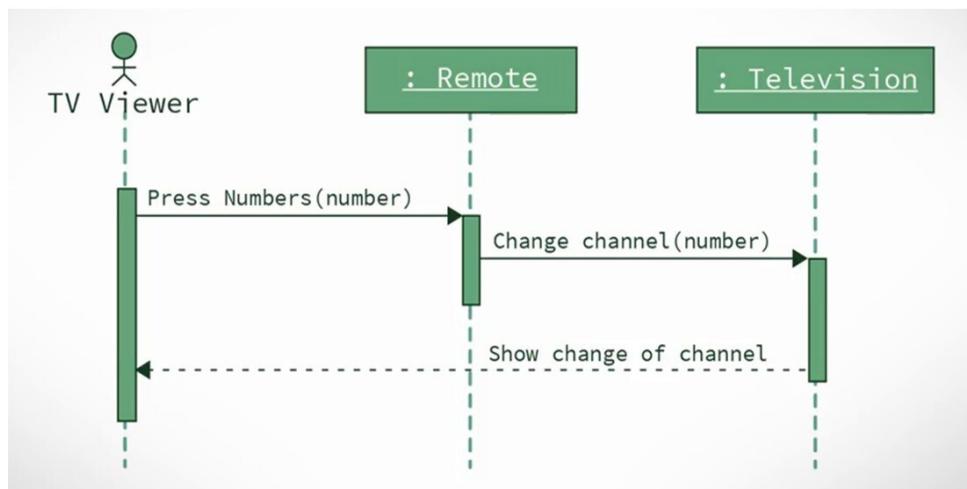
Роль обычно представлена именем класса объекта.

Во-вторых, вы используете вертикальные пунктирные линии, известные как линии жизни, для представления объекта во времени.



И наконец, вы используете стрелки для отображения сообщений, отправленных с одного объекта на другой.

Чтобы сохранить ясность, вы должны рисовать объекты слева направо в последовательности, в которой они взаимодействуют друг с другом.



И если в вашем примере есть люди, которые будут использовать или взаимодействовать с объектами, они, как правило, изображаются фигурой.

Мы называем этих людей действующими лицами.

Здесь, в этом примере, телезрителем является действующее лицо.

В диаграмме последовательности, если один объект отправляет сообщение другому объекту или объектам, мы обозначаем это, вычерчивая сплошную стрелку линии от отправителя к получателю.

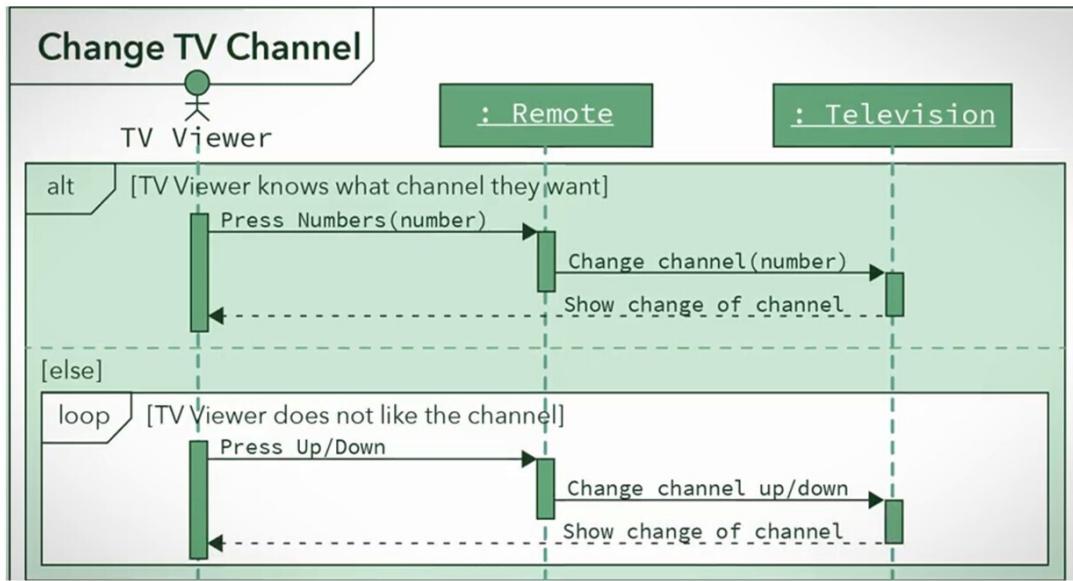
Чтобы возвращать данные, мы используем пунктирную стрелку.

Когда объект активирован, мы обозначаем это на диаграмме последовательности, используя маленькие прямоугольники на линии жизни объектов.

Вы активируете объект всякий раз, когда объект отправляет, получает или ожидает сообщение.

При разработке программного обеспечения диаграммы последовательности могут значительно усложняться.

Вы можете показывать циклы и альтернативные процессы на диаграмме последовательности.



Предположим, что телезритель не уверен, какой канал выбрать, и он хотел бы попробовать каналы, пока не найдет подходящий канал.

Мы можем нарисовать последовательность, как часть альтернативного процесса.

Это последовательность действий, которые произойдут, если условие истинно.

Мы помещаем эту последовательность в поле и маркируем ее как alt, для альтернативы, в верхнем правом углу.

Теперь мне нужно указать, когда эта альтернатива произойдет.

В этом случае последовательность возникает, если телезритель знает, какой канал ему нужен.

И мы обозначим эту альтернативу с помощью строки.

Если телезритель не знает, какой канал он хочет, могут произойти другие последовательности.

Одна из последовательностей заключается в том, что телезритель будет просматривать каналы, пока он не найдет что-то для просмотра.

Поэтому мы нарисуем эту последовательность под предыдущей последовательностью с условием else, что означает, что эта последовательность возникает, если все остальные альтернативы являются ложными.

Однако эта последовательность содержит цикл.

И мы отметим это в прямоугольнике в верхнем правом углу.

Диаграммы последовательностей обычно используются в качестве инструмента дизайна до того, как команда разработчиков начнет программирование.

И эти диаграммы могут помочь вам определить, какие функции вам понадобятся.

Это даже может помочь вам обнаружить проблемы в вашей системе, которые вы не видели раньше.

Задание

Создайте диаграмму последовательности для аэропорта.

Эта диаграмма должна показывать, как взаимодействуют классы системы, когда клиент покупает авиабилет на веб-сайте бронирования.

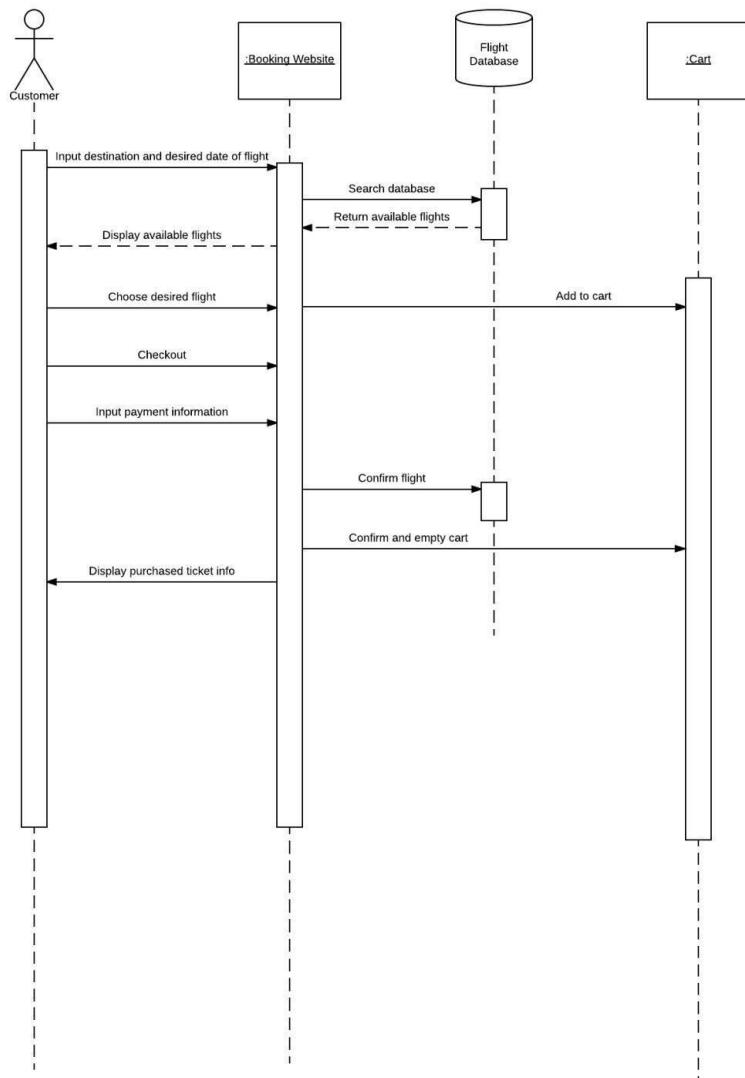
Система должна позволять клиенту осуществлять поиск доступных рейсов из базы данных, введя их местоположение и дату вылета/прибытия.

Веб-сайт будет искать в базе данных и возвращать доступные рейсы для отображения.

Как только клиент выбрал рейс, он добавляется в корзину.

Затем клиент вводит свои платежные данные, и как только все будет завершено, веб-сайт должен подтвердить рейс, освободить корзину и, наконец, отобразить подтверждение билета на рейс.

Бесплатный онлайн-инструмент, который вы можете использовать для создания вашей диаграммы – Lucidchart <https://www.lucidchart.com>.



UML диаграмма состояний

Теперь давайте обсудим UML диаграммы состояний.

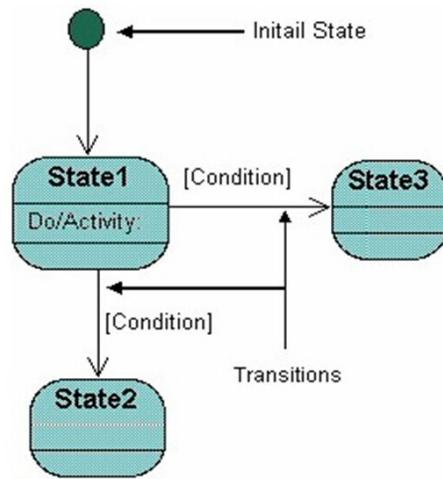
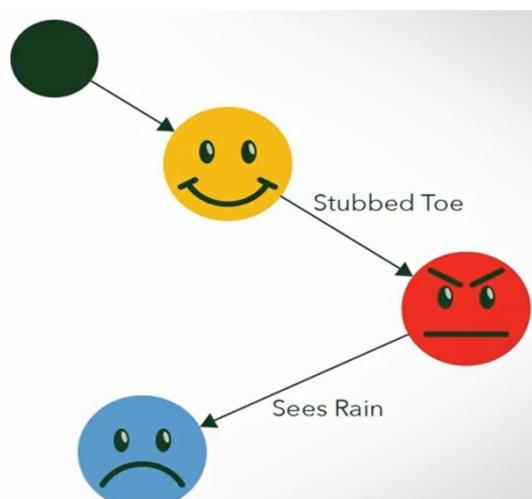


Диаграмма состояния – это способ, который вы можете использовать для описания того, как ваша система себя ведет и реагирует.

Когда происходит событие, вы можете отметить, как система действует или ведет себя. Давайте рассмотрим пример человека.



Предположим, что у человека могут быть три разных эмоциональных состояния, он может быть счастливым, грустным и сердитым.

Диаграмма состояний показывает состояния как узлы.

И в диаграммах состояний стрелки используются для представления событий для перехода из одного состояния в другое.

Диаграмма состояния соответствует состояниям системы или объекта и показывает изменения между состояниями по мере возникновения событий.

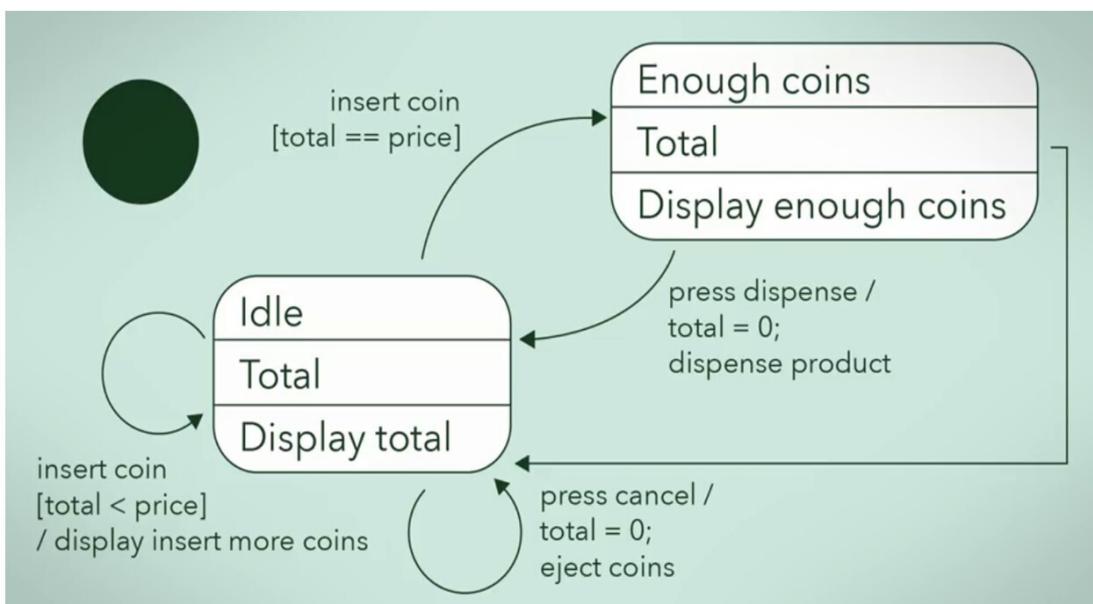
Здесь это упрощенная версия диаграммы состояния человека.

Состояние – это способ существования объекта в определенный момент времени.

И состояние объекта определяется значениями его атрибутов.

Когда объект находится в определенном состоянии, он ведет себя определенным образом или имеет атрибуты, установленные для определенных значений.

Используя диаграммы состояний UML, вы можете выразить разные состояния объектов и то, как эти состояния будут меняться при возникновении события.



Рассмотрим создание диаграммы состояний UML на примере торгового автомата.

Во-первых, мы укажем начало этой диаграммы с помощью заполненного круга.

Каждая диаграмма состояний имеет заполненный круг, указывающий, что является начальным состоянием.

И торговый автомат имеет начальное состояние с именем `idle` или бездействие.

Это когда торговый автомат ожидает ввода монет.

И мы рисуем состояния как закругленные прямоугольники.

Каждое состояние имеет три раздела: имя состояния, переменные состояния и действия.

И каждое состояние должно иметь как минимум имя состояния.

Переменные состояния – это данные, относящиеся к состоянию объекта.

И действия – это действия, выполняемые в определенном состоянии.

Существует три типа действий для каждого состояния, входа, выхода и выполнения.

Действия входа – это действия, которые происходят, когда состояние возникает из другого состояния.

Действия выхода – это действия, которые происходят, когда состояние завершается и переходит в другое состояние.

И действия выполнения – это действия, которые происходят один раз или несколько раз, пока объект находится в определенном состоянии.

Когда торговый автомат входит в состояние бездействия, он всегда отображает общее количество вставленных до этого монет.

Вставка монеты – это событие, которое может изменить состояние торгового автомата.

Предположим, что в состоянии бездействия кто-то вставляет монету, и общая сумма пока меньше цены продукта.

При этом стрелка перехода возвращается в состояние ожидания.

И предположим, что кто-то вставляет монету, и сумма становится равна цене продукта.

И стрелка перехода указывает новое состояние – с именем достаточно монет.

В этом состоянии, если кто-то нажимает кнопку выдачи, торговый автомат должен выдать один из продуктов.

При этом стрелка указывает переход из состояния достаточного количества монет обратно в режим ожидания.

И в любом состоянии, если кто-то нажмет кнопку отмены, автомат должен вернуть все вставленные монеты.

И один элемент диаграммы состояния, который не показан здесь, является завершением.

Завершение представляет собой объект, который уничтожается или процесс, который завершается, и рисуется как круг с заполненным внутри кругом.

Например, в примере с банкоматом вы можете представить завершение, как возврат карточки в конце процесса.

Не все диаграммы имеют завершение, как в случае с торговым автоматом, который работает непрерывно.

Диаграммы состояний полезны для описания поведения системы или одного объекта.

Например, это может помочь вам определить различные события, которые могут возникнуть во время жизни объекта, и как этот объект должен вести себя, когда происходят эти события.

И диаграммы состояний также могут помочь вам найти проблемы в вашей системе.

Например, вы можете обнаружить условие, которое вы не рассматривали ранее.

Задание

Вы должны создать диаграмму состояний UML, представляющую состояния самолета в аэропорту.

Самолет должен проходить через несколько разных состояний.

Когда самолет не используется для полета, он обычно ждет рейс.

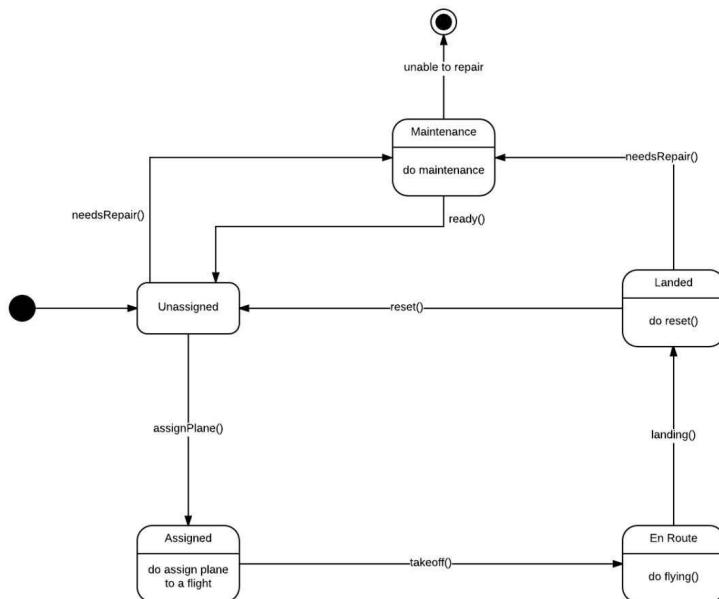
Как только самолет выбран для использования, он назначается на рейс, пока самолет не будет готов к взлете.

Пока самолет находится в воздухе и летит, состояние называется «в пути».

Когда самолет добирается до места назначения, самолет должен перейти в состояние посадки, чтобы аэропорт подготовился к его прибытию.

Наконец, как только самолет успешно приземлился, самолет проверяется, будет ли он готов к назначению на новый рейс или требуется техническое обслуживание.

Если требуется техническое обслуживание, самолет непригоден, и, если механик решает, что самолет не может быть отремонтирован, он удаляется из аэропорта.



Вопросы

Вопрос 1

Какие из следующих терминов используются для описания связывания?

Гибкость +

Частота

Легкость +

Степень +

Открытость

Вопрос 2

Что из следующего является наиболее подходящим?

высокая когезия, свободное связывание +

высокая когезия, тесное связывание

низкая когезия, свободное связывание

низкая когезия, тесное связывание

Вопрос 3

Какие ключевые слова вы можете использовать для скрытия информации в Java?

Защищенный +

Ничего +

Приватный +

Абстрактный

Вопрос 4

Каковы наилучшие способы продвижения концептуальной целостности в программном обеспечении?

Планирование архитектуры системы +

Хорошие комментарии

Регулярные обзоры кода +

Делегирование разработки различных компонентов для разных команд

Вопрос 5

Скрытие информации тесно связано с одним из основных принципов проектирования объектно-ориентированного дизайна. Каким?

Обобщение

Инкапсуляция +

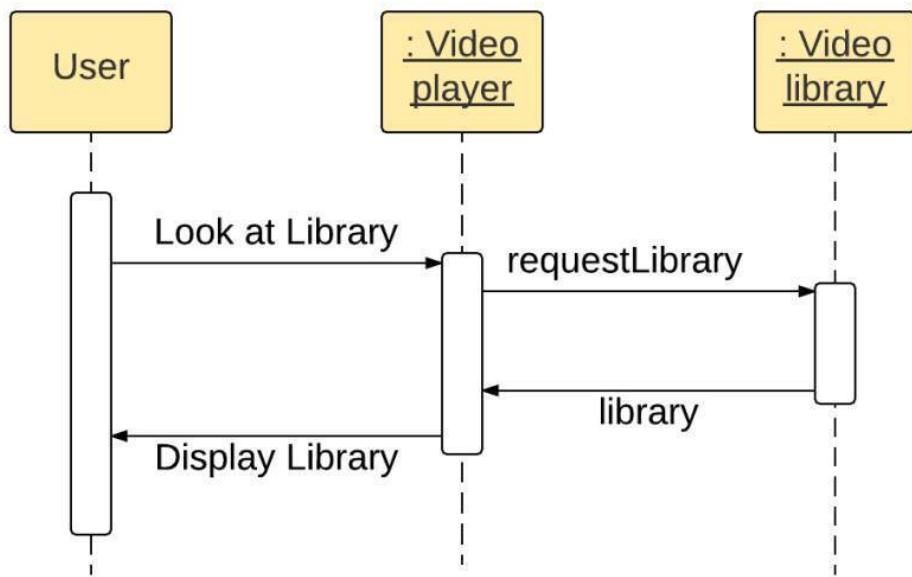
Абстракция

Декомпозиция

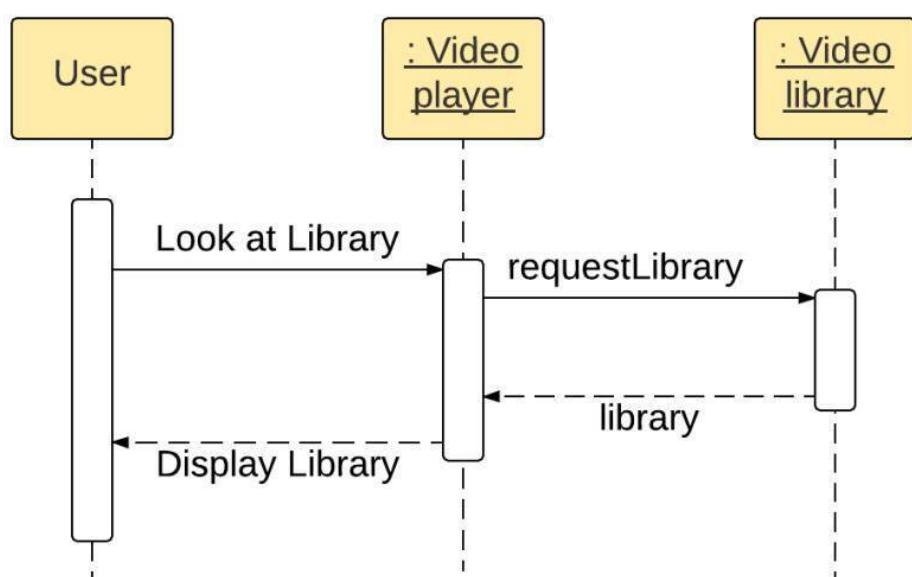
Вопрос 6

Какая из этих диаграмм последовательности верна?

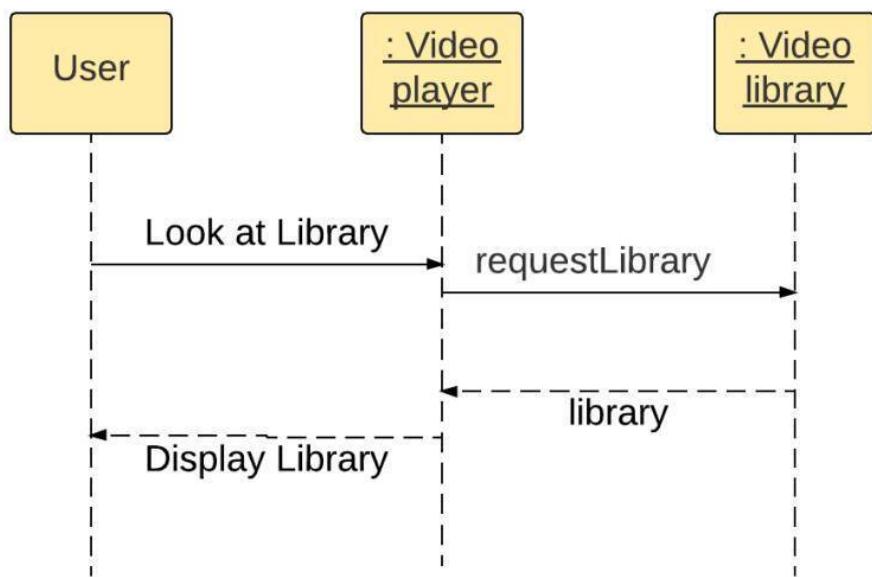
a)



b)



c)



a)

b) +

c)

d)

Вопрос 7

Каковы есть элементы состояния в диаграмме состояний?

Имя состояния +

Переменные состояния +

События

Действия +

Обязанности

Паттерны проектирования

Что такое шаблон проектирования?

Паттерн проектирования - это практическое доказанное решение повторяющейся задачи проектирования.

При разработке программного обеспечения вы часто сталкиваетесь с одной и той же задачей проектирования много раз.

Существует много способов справиться с этими повторяющимися задачами, но со временем некоторые решения предпочтительнее других, потому что они более гибкие или многоразовые.

Шаблон проектирования – это практическое доказанное решение повторяющейся задачи проектирования.

Вы можете использовать ранее разработанные решения, которые часто используют разработчики.

Эти решения не являются теоретическими предложениями, имеющими только академический интерес. Это практические решения, которые используются в промышленном программном обеспечении.

На протяжении многих лет разработчики экспериментировали со множеством различных решений проектирования. И эти шаблоны описывают решения, которые часто создают лучший результат.

В известной книге под названием «Шаблоны проектирования. Элементы многоразового объектно-ориентированного программного обеспечения» авторов Гамма, Хелм, Джонсон и Влиссидес выделены 23 шаблона проектирования.

Команда авторов этой книги известна как «Банда четырёх» Gang of Four, часто сокращается до GoF. Именно эта книга стала причиной роста популярности шаблонов проектирования.

Важно отметить, что не так просто взглянуть на конкретную задачу разработки программного обеспечения и автоматически понять, какой шаблон дизайна использовать.

Существует много шаблонов, и иногда может показаться, что может быть применим один шаблон проектирования, но на самом деле он не является подходящим для данной задачи.

Только благодаря опыту можно научиться лучше судить, какие шаблоны использовать в конкретной ситуации.

Другое замечание о шаблонах проектирования заключается в том, что это не просто конкретный набор исходного кода, который вы запоминаете и помещаете в свое программное обеспечение.

Шаблоны проектирования представляют концепции.

Это знание, которое вы можете применить в своем программном обеспечении для управления своей структурой, чтобы сделать ее более гибкой и повторно используемой.

И наконец, шаблоны проектирования – это словарный запас.

Вместо того, чтобы объяснять детали дизайнерского решения, снова и снова, вы можете упростить дискуссию, используя одно слово, чтобы описать решение.

Знание шаблонов проектирования оставляет меньше места для недопонимания.

Таким образом, шаблон проектирования представляет собой хорошо зарекомендовавшее себя решение для конкретной задачи.

Эти решения основаны на знаниях и опыте опытных разработчиков программного обеспечения.

Например, вы хотите создать класс, для которого должен быть создан только один экземпляр или объект, и что только один объект может использоваться всеми другими классами.

Лучшее решение этой задачи – это шаблон проектирования Singleton.

Каждый шаблон проектирования содержит некоторую спецификацию или набор правил для решения определенной задачи.

При этом шаблоны проектирования – это независимые от языка программирования стратегии для решения общих объектно-ориентированных задач.

Это означает, что шаблон проектирования представляет собой идею, а не конкретную реализацию.

Используя шаблоны проектирования, вы можете сделать свой код более гибким, поддерживаемым и улучшить его повторное использование.

Сама технология java под капотом следует шаблонам проектирования.

Чтобы стать профессиональным разработчиком программного обеспечения, вы должны знать, по крайней мере, некоторые популярные шаблоны проектирования для кодирования задач.

Шаблоны проектирования не гарантируют абсолютное решение проблемы. Они обеспечивают ясность архитектуры системы и возможность создания более совершенной системы.

Когда мы должны использовать шаблоны проектирования?

Мы должны использовать шаблоны проектирования на этапе анализа и сбора требований жизненного цикла разработки программного обеспечения.

Шаблоны проектирования облегчают этот этап, предоставляя информацию, основанную на предыдущем практическом опыте.

Шаблоны проектирования разделяют на две части – это шаблоны проектирования платформы Java SE и шаблоны проектирования платформы JEE.

Мы сосредоточимся на шаблонах проектирования платформы Java SE.

В 1994 году банда четырех опубликовали книгу «Шаблоны проектирования – элементы многоразового объектно-ориентированного программного обеспечения», которая представила концепцию шаблонов проектирования в разработке программного обеспечения.

Следуя этим авторам шаблоны проектирования в основном базируются на следующих принципах объектно-ориентированного проектирования.

Программировать на уровне интерфейса, а не на уровне реализации, и использовать композицию объектов вместо наследования.

Программировать на уровне интерфейса, а не на уровне реализации, и использовать композицию объектов вместо наследования.

У манипулирования объектами строго через интерфейс абстрактного класса есть два преимущества – пользователю не нужно иметь информации о конкретных типах объектов, которыми он пользуется, при условии, что все они имеют ожидаемый клиентом интерфейс и пользователю необязательно "знать" о классах, с помощью которых реализованы объекты.

Клиенту известно только об абстрактном классе, определяющем интерфейс.

Интерфейс дает гибкость: если не устраивает одна его реализация, мы сможем легко переключиться на другую реализацию, написав класс, реализующий этот интерфейс.

Основная цель ООП заключается в том, чтобы код создаваемый в одном месте можно было использовать в разных местах программы.

При этом сам код не дублируется.

Для реализации этой цели у программиста есть два мощных метода – наследование и композиция.

И у того и у другого метода есть как достоинства, так и недостатки.

Самый большой недостаток наследования заключается в том, что оно легко нарушает один из базовых принципов ООП – инкапсуляцию.

Инкапсуляция, это когда мы обращаемся с объектами как с единой сущностью, а не как с набором отдельных полей и методов, тем самым скрываем и защищаем реализацию класса. Если клиентский код не знает ничего, кроме публичного интерфейса, он не может зависеть от деталей реализации.

Еще раз, недостаток наследования заключается в том, что оно легко нарушает один из базовых принципов ООП – инкапсуляцию.

Это связано с тем, что фактически родительский класс определяет поведение дочернего класса, а это значит, что даже незначительное изменение в родительском классе может сильно сказаться на поведении класса-потомка.

Плюс ко всему, повторное использование кода сильно затрудняется, если реализация родителя содержит решения, несовместимые с задачами потомка.

Чтобы выйти из такой ситуации, проводится глубокий рефакторинг кода, а это не всегда возможно.

На практике, чтобы избежать зависимостей от реализации, лучше наследовать абстрактные классы или интерфейсы. Тогда класс-потомок может сам определить каким образом реализовать свою работу.

В противовес наследованию, часто используется другой метод – композиция.

Композиция объектов строится динамически за счет связывания одного объекта с другими.

При таком подходе классы используются в соответствии с их интерфейсом.

Что не нарушает инкапсуляцию.

Использование единого интерфейса позволяет в дополнение к инкапсуляции получить преимущества полиморфизма.

Т.е. во время выполнения программы можно один объект заменить другим, при условии, что у него такой же интерфейс.

Композиция, это когда поле в классе у нас имеет тип Класс, оно может содержать ссылку на другой объект этого класса, создавая таким образом связь между двумя объектами.

Композиция – это когда один объект предоставляет другому свою функциональность частично или полностью.

Наследование применяется, если оба класса, родитель и потомок, из одной предметной области, наследник является корректным подтипов родителя, код родителя необходим либо хорошо подходит для наследника, и наследник в основном добавляет логику.

Во всех остальных случаях используется композиция.

Есть много шаблонов дизайна, которые вы можете найти в книгах и в Интернете.

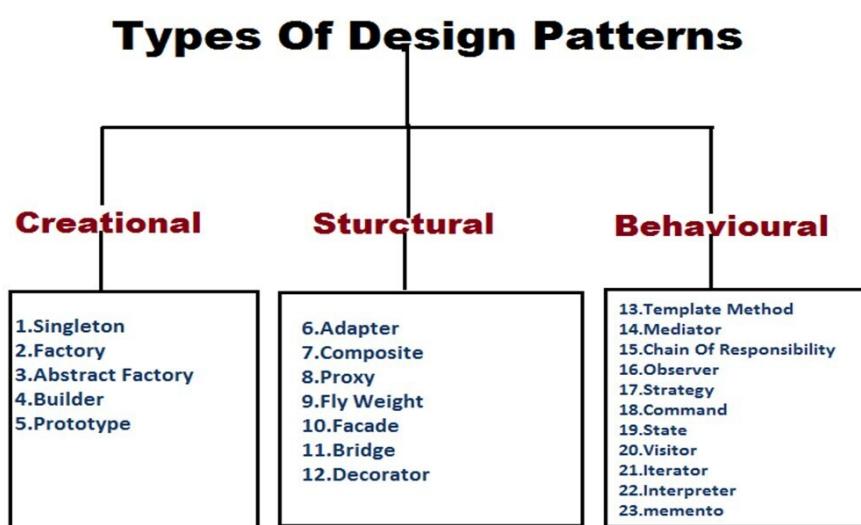
Одна из самых известных книг – это книга «Банды четырех».

Книга «Банда четырех» обобщила опыт разработчиков.

Каждый автор писал свои собственные приложения, и при этом обнаружил шаблоны, возникающие в их дизайнерских решениях.

Как группа, они решили формализовать эти шаблоны.

И эти шаблоны подразделяются на категории.



Одна из основных категорий шаблонов включает в себя Creational Patterns или порождающие шаблоны.

Creational Patterns решают, как вы обрабатываете создание новых объектов.

Существует несколько различных шаблонов, основанных на создании и клонировании объектов.

Например, если вы создаете объект, похожий на существующий, вместо создания экземпляра нового объекта, вы можете клонировать существующий объект.

Другая категория шаблонов, используемая «Бандой четырех», включает в себя Структурные шаблоны.

Структурные шаблоны, описывают, как объекты связаны друг с другом.

Ранее мы рассматривали основные принципы проектирования, такие как декомпозиция и обобщение, и как они выражены в диаграммах классов UML отношениями ассоциации, агрегации, композиции, наследования и интерфейса.

Существует много разных способов структурирования объектов в зависимости от отношений между ними.

Структуры не только описывают, как разные объекты имеют отношения между собой, но также и как подклассы и классы взаимодействуют через наследование.

Структурные шаблоны используют эти отношения и описывают, как они должны работать для достижения определенной цели дизайна.

И еще одна категория шаблонов включает в себя поведенческие шаблоны.

Эти шаблоны фокусируются на том, как объекты распределяют работу.

Они описывают, как каждый объект выполняет одну когезивную функцию.

Поведенческие шаблоны также сосредоточены на том, как независимые объекты работают для достижения общей цели.

Все эти категории не являются четко определенными.

Некоторые шаблоны имеют элементы, которые могут быть отнесены к разным категориям.

«Банда четырех» просто использует эти категории, чтобы упорядочить и описать шаблоны в своей книге.

И мы начнем рассмотрение паттернов проектирования с группы паттернов, которая называется Порождающие шаблоны или Creational Design Pattern.

Factory Method Pattern

Первый шаблон, это Factory Method Pattern или Фабричный метод, или Виртуальный конструктор.

Точно так же, как фабрика в реальном мире, целью этих фабрик в объектно-ориентированном программировании является создание объектов.

Использование фабрик упрощает обслуживание и изменение программного обеспечения, так как создание объектов происходит на фабриках.

Методы, которые используют эти фабрики, могут быть сфокусированы на другом поведении.

```
Knife orderKnife(String knifeType) {  
    Knife knife;  
  
    // create Knife object - concrete instantiation  
    if (knifeType.equals("steak")) {  
        knife = new SteakKnife();  
    } else if (knifeType.equals("chefs")) {  
        knife = new ChefsKnife();  
    }  
  
    // prepare the Knife  
    knife.sharpen();  
    knife.polish();  
    knife.package();  
  
    return knife;  
}
```

Представьте, что ваше программное обеспечение реализует интернет-магазин по продаже ножей и что вы хотите создавать объекты для продажи в этом магазине.

Существует много разных видов ножей, но давайте начнем с SteakKnife и ChefsKnife.

И у вас есть суперкласс ножей с подклассами SteakKnife и ChefsKnife.

Вы пишете метод заказа ножа, который сначала создаст один из этих объектов ножа, а затем подготовит его к отправке.

Предположим, что подготовка ножа заключается в его заточке, полировке и упаковке.

Здесь сначала метод объявляет переменную ножа, которая будет ссылаться на объект ножа, который будет создан.

Условное выражение определяет, экземпляр какого подкласса ножа фактически создается.

Следующим шагом будет вызов методов, которые являются общими для различных типов ножей, это заточка, полировка и упаковка.

Этим методам не важно, какой тип ножа создан, все, что им нужно, – это нож для работы.

И представьте себе, что ваш магазин добавляет все больше и больше типов ножей, поскольку его продажи улучшаются.

Новые подклассы добавляются по мере необходимости, хлебный нож, нож для резки и т. д.

В этом примере список условных выражений растет и растет при добавлении новых типов ножей.

И обратите внимание, что то, что мы делаем с ножом после его создания, не меняется.

Все эти ножи необходимо затачивать, полировать и упаковывать.

И все это становится довольно сложным.

Что, если вместо того, чтобы создавать ножи в магазине, вы можете сделать их где-то в другом месте?

Как и в реальном мире, объекты обычно производятся на фабриках.

Мы можем создать объект фабрики, роль которого заключается в создании объектов определенных типов.

```
public class KnifeFactory {  
    public Knife createKnife(String knifeType) {  
        Knife knife = null;  
  
        // create Knife object  
        if (knifeType.equals("steak")) {  
            knife = new SteakKnife();  
        } else if (knifeType.equals("chefs")) {  
            knife = new ChefsKnife();  
        }  
  
        return knife;  
    }  
}
```

Таким образом, заточка, полировка и упаковка останутся там, где они находятся при заказе ножа.

Но вы делегируете ответственность за создание объекта другому объекту – KnifeFactory.

Вы переместите код для определения того, какой тип ножа нужно создать, а также код для определения того, какой подкласс ножа нужно использовать в классе этой фабрики.

Некоторый другой код создаст объект KnifeFactory, но как только это будет сделано, вы можете использовать его для создания ножей определенных типов.

```
public class KnifeStore {  
    private KnifeFactory factory;  
    // require a KnifeFactory object to be passed to  
    // this constructor:  
    public KnifeStore(KnifeFactory factory) {  
        this.factory = factory;  
    }  
    public Knife orderKnife(String knifeType) {  
        Knife knife;  
        // use the create method in the factory  
        knife = factory.createKnife(knifeType);  
        // prepare the Knife  
        knife.sharpen();  
        knife.polish();  
        knife.package();  
        return knife;  
    }  
}
```

В этом примере используется объект KnifeFactory, который передается в конструктор класса KnifeStore.

Метод orderKnife очень похож на предыдущий.

Однако вместо создания конкретного экземпляра он делегирует свою задачу объекту фабрики.

Что вы получили? Прежде всего, KnifeStore и его метод orderKnife могут быть не единственным клиентом вашего KnifeFactory.

Другие клиенты также могут использовать фабрику ножей для создания ножей для других целей.

Может быть, есть метод для оптовых заказов или проверки качества.

И так как все фактическое создание ножа происходит в KnifeFactory, вы можете просто добавить новые типы ножей в свой KnifeFactory без изменения кода клиента.

Если есть несколько клиентов, которые хотят создать один и тот же набор классов, тогда, используя объект фабрики, вы удаляете избыточный код и упрощаете его модификацию.

Хотя это полезная техника, Factory Object на самом деле не является одним из шаблонов банды четырех.

Они описывают шаблон Factory Method.

Шаблон фабрики метода создает конкретные типы объектов по-другому.

Вместо использования отдельного объекта – объекта фабрики для создания объектов, Factory Method использует отдельный метод в том же классе для создания объектов.

Целью шаблона Factory Method является определение интерфейса для создания объектов, но пусть подклассы решают, экземпляр какого класса необходимо создать.

```
public abstract class KnifeStore {  
  
    public Knife orderKnife(String knifeType) {  
        Knife knife;  
  
        // now creating a knife is a method in the class  
        knife = createKnife(knifeType);  
  
        // this is still the same as before!  
        knife.sharpen();  
        knife.polish();  
        knife.package();  
  
        return knife;  
    }  
  
    abstract Knife createKnife(String knifeType);  
  
}
```

Как это выглядит?

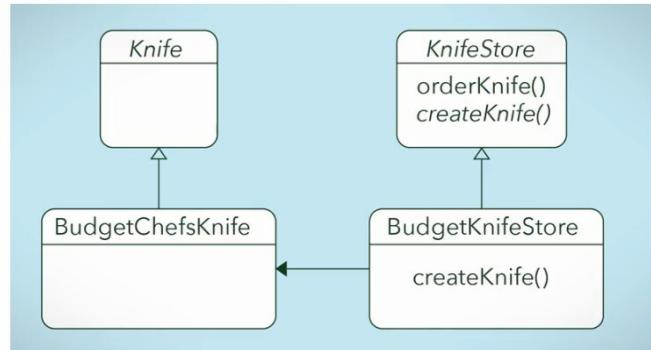
Сначала посмотрим на суперкласс KnifeStore.

Прежде всего, KnifeStore теперь абстрактный.

Это означает, что мы не можем создать экземпляр KnifeStore.

Вместо этого вам понадобятся подклассы KnifeStore, такие как Budget KnifeStore или Quality KnifeStore.

```
public BudgetKnifeStore extends KnifeStore {  
  
    // up to any subclass of KnifeStore to define this method  
    Knife createKnife(String knifeType) {  
        if (knifeType.equals("steak")) {  
            return new BudgetSteakKnife();  
        } else if (knifeType.equals("chefs")) {  
            return new BudgetChefsKnife();  
        }  
        //... more types  
        else return null;  
    }  
}
```



Эти подклассы наследуют один и тот же метод создания ножа.

Он объявлен в суперклассе, но он абстрактный и пустой.

Мы оставили метод фабрики пустым и назвали его абстрактным, потому что мы хотим, чтобы этот метод определялся подклассами.

Всякий раз, когда определяется подкласс KnifeStore, он должен определить этот метод createKnife.

BudgetKnifeStore имеет собственный метод создания объектов ножа.
У другого магазина ножей, такого как QualityKnifeStore, будет другой метод фабрики.
И вы не можете определить новый подкласс KnifeStore без предоставления этого метода.
И потому что BudgetKnifeStore является подклассом KnifeStore, он наследует метод orderKnife.

Все подклассы KnifeStore наследуют метод orderKnife.

Однако каждый подкласс должен определить свой собственный метод создания ножа.

Здесь показана диаграмма класса UML.

И диаграмма классов UML дает важную информацию.

Прежде всего, нож и магазин ножей выделены курсивом для обозначения что это абстрактные классы.

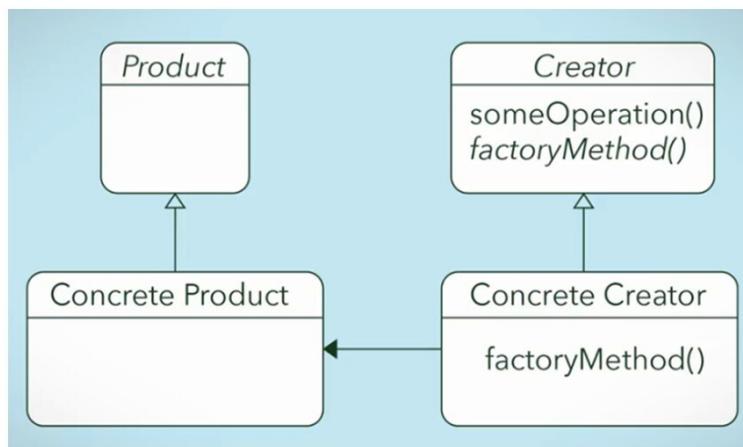
Вы не можете создавать эти классы напрямую, вы должны определить их подклассы.

Диаграмма UML показывает только один подкласс ножа, но может быть и больше.

И у нас также может быть больше одного магазина KnifeStore, которые предлагают другие типы ножей.

Это ядро шаблона проектирования Factory Method.

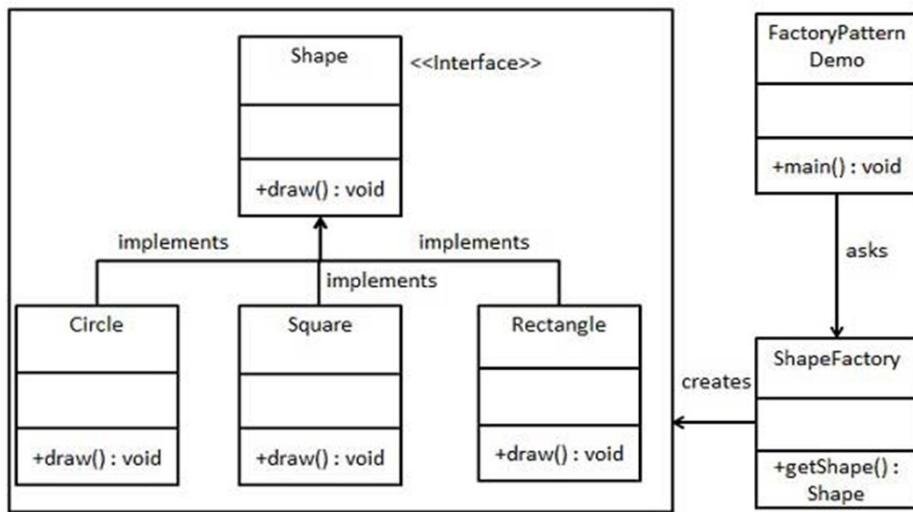
Шаблон Factory Method всегда следует одной и той же общей структуре.



У нас есть класс абстрактного творца.

Этот класс содержит методы, которые обеспечивают обобщение.

И чтобы каждый конкретный класс создателя был обязан предоставить фабричный метод создания конкретного продукта, конкретный создатель является подклассом абстрактного создателя.



Возвращаясь к шаблону Factory Object, в шаблоне Factory Object мы создаем объект, не показывая логику создания клиенту, и ссылаемся на вновь созданный объект, используя общий интерфейс.

Смотрите, здесь у нас есть интерфейс Shape и его реализации – Rectangle, Square и Circle.

Чтобы создать один из этих объектов, мы не создаем экземпляр конкретного класса, а передаем в метод getShape класса ShapeFactory просто тип объекта, который мы хотим получить.

А уже метод getShape создает экземпляр конкретного класса, и мы вызываем для созданного объекта метод интерфейса, ничего не зная о деталях его реализации.

Таким образом, этот шаблон скрывает от нас реализацию интерфейса, и все что нам нужно знать, это интерфейс и тип нужного нам объекта.

При этом заранее мы можем не знать, какой тип объекта нам будет нужен.

Например, тип объекта будет вводить пользователь в уже работающем приложении.

И этот ввод будет в виде строки передаваться в метод getShape класса ShapeFactory.

Это и есть шаблон Factory Object.

```

public interface Shape {
    void draw();
}

public class Rectangle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}

public class Square implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}

public class Circle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}

```

Здесь у нас интерфейс Shape, его метод draw, который мы хотим использовать, и его реализации – Rectangle, Square и Circle.

```

public class ShapeFactory {

    //use getShape method to get object of type shape
    public Shape getShape(String shapeType){
        if(shapeType == null){
            return null;
        }
        if(shapeType.equalsIgnoreCase("CIRCLE")){
            return new Circle();
        }
        else if(shapeType.equalsIgnoreCase("RECTANGLE")){
            return new Rectangle();
        }
        else if(shapeType.equalsIgnoreCase("SQUARE")){
            return new Square();
        }
        return null;
    }
}

```

Затем у нас есть фабрика ShapeFactory с фабричным методом getShape, который принимает строковое описание типа объекта и возвращает объект нужного нам типа.

```

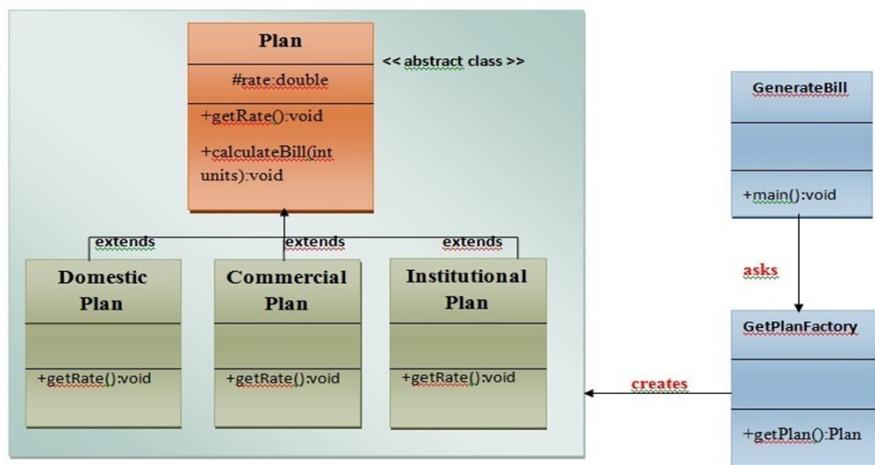
public class FactoryPatternDemo {
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();

        //get an object of Circle and call its draw method.
        Shape shape = shapeFactory.getShape("CIRCLE");

        //call draw method of Circle
        shape.draw();
    }
}

```

И наконец, в пользовательском коде, мы создаем экземпляр фабрики, вызываем его метод `getShape`, и обращаемся к полученному объекту как к интерфейсу, вызывая метод `draw` интерфейса.

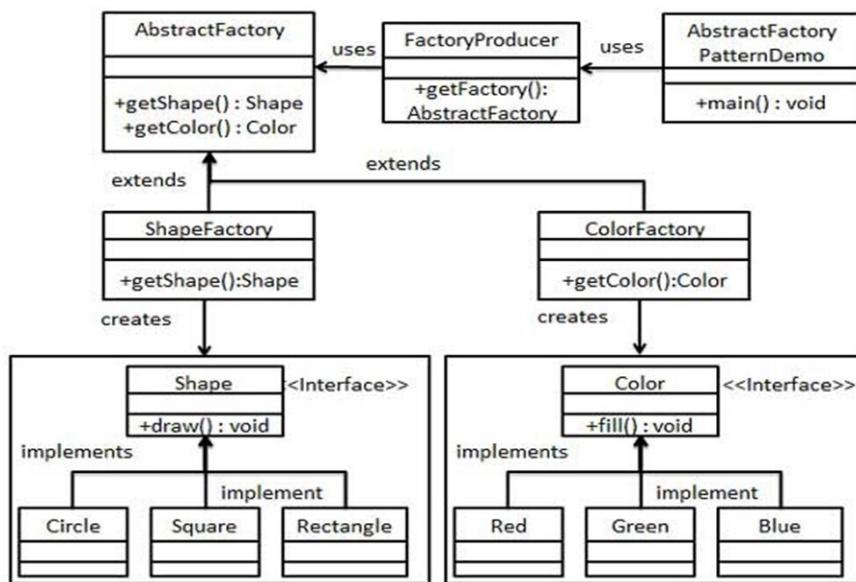


Шаблон Factory Object используется, в общем и целом, когда код заранее не знает, какая реализация интерфейса будет использоваться, но при этом нужно вызвать метод объекта класса, реализующего интерфейс.

Здесь мы не знаем заранее какой план нам понадобится, но мы можем использовать методы `getRate` и `calculateBill`, создав фабричный метод `getPlan`.

Abstract Factory Pattern

Следующий шаблон, это Абстрактная фабрика.



Шаблон Абстрактная фабрика представляет суперфабрику, которая создает другие фабрики.

Эта фабрика также называется фабрикой фабрик.

В шаблоне Abstract Factory, интерфейс или абстрактный класс отвечает за создание фабрики связанных объектов без явного указания их классов.

Это означает, что Abstract Factory позволяет классу возвращать фабрику классов.

В этом примере у нас есть связанные объекты формы и цвета.

Для их использования мы создаем абстрактную фабрику с двумя методами getColor, который возвращает объект цвета, и getShape, который возвращает объект формы.

Затем мы создаем две реализации этой абстрактной фабрики – фабрику ShapeFactory, в которой реализуем метод getShape, возвращающий конкретную реализацию формы, и фабрику ColorFactory, в которой реализуем метод getColor, возвращающий конкретную реализацию цвета.

И создаем класс FactoryProducer с методом getFactory, который возвращает конкретную фабрику.

Теперь мы можем в клиентском коде создать FactoryProducer, получить нужную нам фабрику, и затем получить нужный нам объект цвета или формы.

```
public interface Shape {
    void draw();
}

public class Rectangle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}

public class Square implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}

public class Circle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}
```

У нас есть интерфейс Shape и его реализации Rectangle, Square и Circle.

```
public interface Color {
    void fill();
}

public class Red implements Color {

    @Override
    public void fill() {
        System.out.println("Inside Red::fill() method.");
    }
}

public class Green implements Color {

    @Override
    public void fill() {
        System.out.println("Inside Green::fill() method.");
    }
}

public class Blue implements Color {

    @Override
    public void fill() {
        System.out.println("Inside Blue::fill() method.");
    }
}
```

У нас есть интерфейс Color и его реализации Red, Green и Blue.

```

public class ShapeFactory extends AbstractFactory {

    @Override
    public Shape getShape(String shapeType){

        if(shapeType == null){
            return null;
        }

        if(shapeType.equalsIgnoreCase("CIRCLE")){
            return new Circle();
        }

        }else if(shapeType.equalsIgnoreCase("RECTANGLE")){
            return new Rectangle();
        }

        }else if(shapeType.equalsIgnoreCase("SQUARE")){
            return new Square();
        }

        return null;
    }

    @Override
    Color getColor(String color) {
        return null;
    }
}

```

```

public abstract class AbstractFactory {
    abstract Color getColor(String color);
    abstract Shape getShape(String shape);
}

```

У нас есть абстрактная фабрика и ее реализация ShapeFactory с реализацией метода getShape.

```

public class ColorFactory extends AbstractFactory {

    @Override
    public Shape getShape(String shapeType){
        return null;
    }

    @Override
    Color getColor(String color) {

        if(color == null){
            return null;
        }

        if(color.equalsIgnoreCase("RED")){
            return new Red();
        }

        }else if(color.equalsIgnoreCase("GREEN")){
            return new Green();
        }

        }else if(color.equalsIgnoreCase("BLUE")){
            return new Blue();
        }

        return null;
    }
}

```

```

public abstract class AbstractFactory {
    abstract Color getColor(String color);
    abstract Shape getShape(String shape);
}

```

И реализация абстрактной фабрики ColorFactory с реализацией метода getColor.

```

public class AbstractFactoryPatternDemo {
    public static void main(String[] args) {

        //get shape factory
        AbstractFactory shapeFactory =
        FactoryProducer.getFactory("SHAPE");

        //get an object of Shape Circle
        Shape shape = shapeFactory.getShape("CIRCLE");

        //call draw method of Shape Circle
        shape.draw();
        AbstractFactory colorFactory =
        FactoryProducer.getFactory("COLOR");

        //get an object of Color Red
        Color color = colorFactory.getColor("RED");

        //call fill method of Red
        color.fill();
    }
}

```

И у нас есть класс FactoryProducer, с помощью которого теперь мы в коде получаем одну фабрику, с помощью которой получаем круг, и получаем другую фабрику, с помощью которой получаем цвет.

Это фабрика фабрик.

Преимущество шаблона абстрактной фабрики заключается в том, что шаблон Abstract Factory изолирует клиентский код от конкретных реализаций классов.

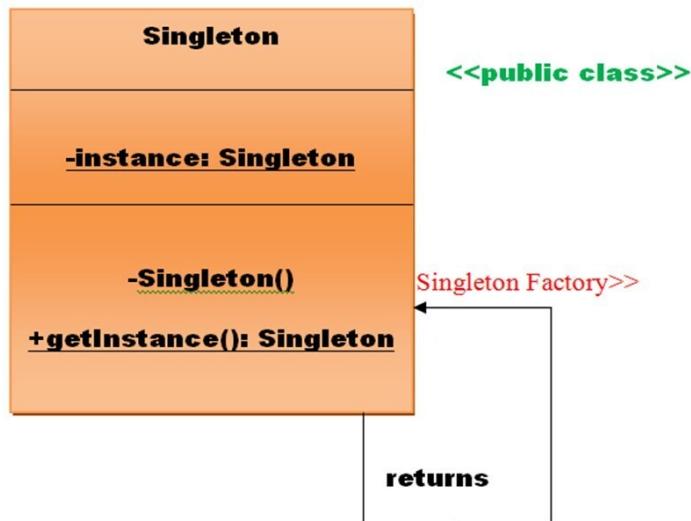
Это облегчает обмен семейств объектов и способствует согласованности между объектами.

Шаблон абстрактной фабрики используется, когда система должна быть независимой от того, как ее объект создается, составляется и представляется.

При использовании семейства связанных объектов вместе, и, если вы хотите предоставить библиотеку объектов, которые не показывают своих реализаций, а только представляют свои интерфейсы.

При этом система может быть настроена с одним объектом из нескольких семейств объектов.

Singleton Pattern



Шаблон Singleton является одним из простейших шаблонов проектирования на Java. Этот шаблон представляет один класс, который несет ответственность за создание объекта, гарантируя, что создается только один объект.

И этот класс предоставляет возможность доступа к его единственному объекту, к которому можно получить доступ напрямую, без необходимости создавать экземпляр объекта класса.

Другими словами, класс должен гарантировать, что только один экземпляр должен быть создан, и только один объект может использоваться всеми другими классами.

Синглтон – это шаблон создания, который описывает способ создания объекта.

Как следует из названия, шаблон дизайна Синглтон – это нечто, имеющее только одно.

Но что это значит? И зачем нам это нужно?

Предположим, вы работаете над мобильным приложением, в котором пользователь может играть в покер.

И в этой игре есть класс предпочтений, определяющий, где хранятся все настройки для пользователя.

Эти предпочтения включают в себя визуальные элементы, такие как цвет игровой поверхности и дизайн карт.

И представьте, что существует более одного экземпляра класса предпочтений.

Когда пользователь устанавливает свои предпочтения, какой объект предпочтений сохранит их.

И когда игра отображается, какой объект предпочтений используется для определения свойств игры?

Как вы можете видеть, наличие другого объекта предпочтений не имеет смысла и может привести к конфликтам или несоответствиям.

Существуют две формы шаблона Singleton.

Раннее создание – это создание экземпляра во время загрузки.

И отложенное создание – это создание экземпляра, когда потребуется.

Преимущество шаблона Singleton в том, что его применение уменьшает использование памяти, так как объект не создается при каждом запросе.

Только один экземпляр класса повторно используется снова и снова.

Шаблон Singleton в основном используется в многопоточных приложениях и приложениях баз данных.

Он используется для ведения журнала, кэширования, пулов потоков, настроек конфигурации и т. д.

```
public class ProductionHouse {  
  
    private static ProductionHouse productionHouse = new ProductionHouse();  
  
    private ProductionHouse(){  
    }  
    public static ProductionHouse getInstance(){  
        return productionHouse;  
    }  
    public void doSomething(){  
        //write your code  
    }  
}
```

Это пример формы шаблона Singleton Раннее Создание.

Здесь Singleton создается во время загрузки класса и сохраняется в приватной статической переменной экземпляра. Метод getInstance обеспечивает извлечение экземпляра.

Таким образом, чтобы создать класс singleton, нам нужно иметь статический член класса, приватный конструктор и статический фабричный метод.

Статический член класса расходует память только один раз из-за статики, он содержит экземпляр класса Singleton.

Приватный конструктор предотвращает создание экземпляра класса Singleton вне класса.

Статический фабричный метод предоставляет глобальную точку доступа к объекту Singleton и возвращает экземпляр вызывающему.

Если Singleton класс загружается двумя загрузчиками классов, будет создано два экземпляра класса, по одному для каждого загрузчика классов.

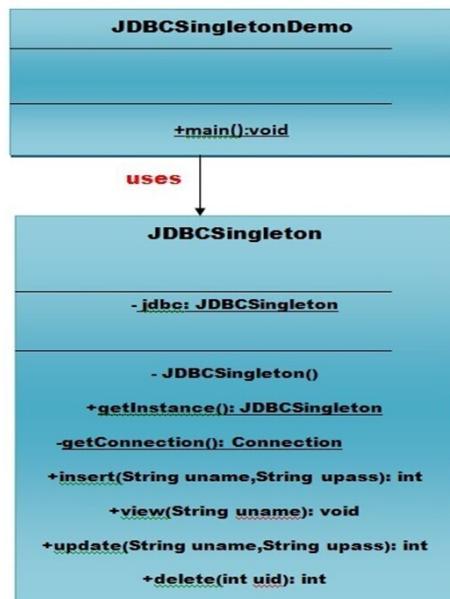
```
public class ProductionHouse {  
  
    private static ProductionHouse productionHouse = null;  
  
    private ProductionHouse() {  
    }  
  
    public static synchronized ProductionHouse getInstance() {  
        if(productionHouse == null) {  
            productionHouse = new ProductionHouse();  
        }  
        return productionHouse;  
    }  
    public void doSomething(){  
        //write your code  
    }  
}
```

Это пример формы шаблона Singleton Отложенное Создание.

Здесь экземпляр объекта создается в первый момент доступа, и это делается потокобезопасно, чтобы предотвратить одновременное создание нескольких экземпляров из разных потоков.

Недостатком этого метода является более низкая производительность, так как метод синхронизирован.

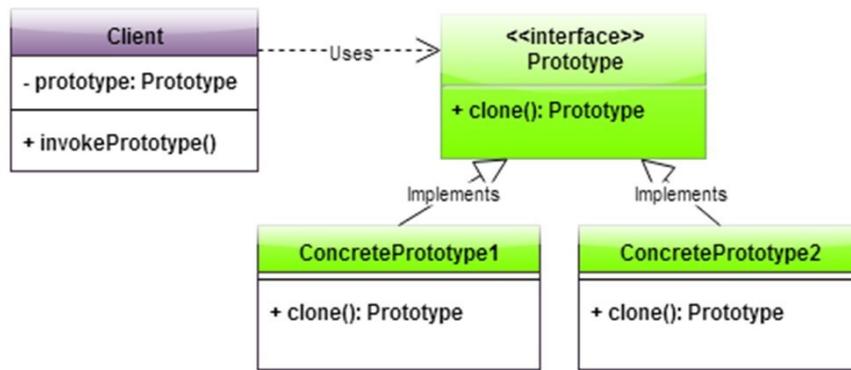
Метод getInstance синхронизируется, и экземпляр создается только в том случае, если он равен нулю.



Пример использования шаблона Singleton – это создание единственного объекта соединения с базой данных, с помощью которого можно работать с данными базы данных.

Prototype Pattern

Следующий шаблон, это Прототип.



Применение шаблона Prototype обеспечивает клонирование существующего объекта вместо создания нового.

Этот шаблон содержит реализацию интерфейса прототипа, который создает клон текущего объекта.

Этот шаблон применяется, если затраты на создание нового объекта дорогостоящие и ресурсоемкие, например, объект должен быть создан в результате ресурсоемкой операции с базой данных.

В этом случае мы можем кэшировать объект, и возвращать его клон при следующем запросе, и обновлять базу данных по мере необходимости, тем самым уменьшая количество запросов к базе данных.

Также если объект является сложным и имеет множество полей, отвечающих за состояние – а пользователю важно только ограничено их количество, тогда с помощью прототипа можно создавать объекты-копии и менять только те поля, которые важны для клиента.

Таким образом, в этом шаблоне проектирования при запуске создается экземпляр объекта (т. е. прототип), а затем всякий раз, когда требуется новый экземпляр, этот прототип кlonируется, чтобы получить другой экземпляр.

```

public interface Prototype {
    public Prototype doClone();
}

public class Person implements Prototype {
    String name;
    String location;

    public Person(String name, String loc) {
        this.name = name;
        this.location=loc;
    }

    @Override
    public Prototype doClone() {
        return new Person(name, location);
    }

    public void setName(String name) {
        this.name = name;
    }
}

public class Dog implements Prototype {
    String sound;
    String location;

    public Dog(String sound, String loc) {
        this.sound = sound;
        this.location=loc;
    }

    @Override
    public Prototype doClone() {
        return new Dog(sound, location);
    }

    public void setSound(String sound) {
        this.sound = sound;
    }
}

```

В этом примере у нас есть интерфейс прототипа и два класса, которые этот интерфейс реализуют.

Предположим, что создание поля `location` является ресурсозатратным.

Тогда мы можем создать по экземпляру данных классов, а затем при следующем запросе их клонировать, и переназначить их поля `name` и `sound`.

Тем самым избежав новое создание поля `location`, используя уже существующее его значение.

Этот шаблон используется интерфейсом `Cloneable` в Java.

```

public abstract class Example implements Cloneable {

    ...

    public Object clone() {
        Object clone = null;

        try {
            clone = super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }

        return clone;
    }
}

```

`Cloneable` реализуется как интерфейс маркер, чтобы показать, какие объекты могут быть клонированы, так как класс `Object` предоставляет защищенный метод `clone`.

Builder Design Pattern

Следующий шаблон, это Строитель.

Шаблон Builder создает сложный объект с использованием простых объектов, используя пошаговый подход.

Класс Builder строит конечный объект шаг за шагом, и он не зависит от других объектов.

Этот шаблон используется главным образом, когда объект не может быть создан за один шаг из-за своей сложности.

```
public class Account {  
  
    private final String userId;  
    private final String token;  
  
    public Account(String token, String userId) {  
        this.token = token;  
        this.userId = userId;  
    }  
  
    public String getUserId() {  
        return userId;  
    }  
  
    public String getToken() {  
        return token;  
    }  
}
```

Предположим у нас есть структура Account и в ней не два поля, как показано здесь, а 20 полей.

Так что для создания экземпляра этой структуры нужно указать в конструкторе 20 параметров и при этом не ошибиться в их типе и порядке.

Поэтому здесь мы применим шаблон строитель.

```

public class Builder {

    public class Account {
        private String userId;
        private String token;

        private Account() {
            // private constructor
        }

        public String getUserId() {
            return userId;
        }

        public String getToken() {
            return token;
        }

        public static Builder newBuilder() {
            return new Account().new Builder();
        }
    }

    private Builder() {
        // private constructor
    }

    public Builder setId(String userId) {
        Account.this.userId = userId;
        return this;
    }

    public Builder setToken(String token) {
        Account.this.token = token;
        return this;
    }

    public Account build() {
        return Account.this;
    }
}

```

В классе Account мы делаем приватный конструктор без параметров и добавляем статический метод newBuilder, возвращающий экземпляр класса Builder.

Затем мы можем пошагово применять к объекту Builder методы, устанавливающие значения полей объекта Account.

И в конце мы вызываем метод build, который возвратит нам собранный объект Account.

В этом примере используется внутренний строитель.

```

public static class Builder {

    public class Account {
        private final String userId;
        private final String token;

        public Account(String userId, String token) {
            this.userId = userId;
            this.token = token;
        }

        public String getUserId() {
            return userId;
        }

        public String getToken() {
            return token;
        }
    }

    private String userId;
    private String token;

    public Builder setId(String userId) {
        this.userId = userId;
        return this;
    }

    public Builder setToken(String token) {
        this.token = token;
        return this;
    }

    public Account build() {
        return new Account(userId, token);
    }
}

```

А это код внешнего строителя, в методе build которого создается экземпляр класса Account с помощью его сложного конструктора.

Structural design patterns. Adapter Pattern

Следующая группа шаблонов, которые мы рассмотрим, это Структурные шаблоны.

Структурные шаблоны проектирования связаны с тем, как классы и объекты могут быть скомпонованы, чтобы сформировать более крупные структуры.

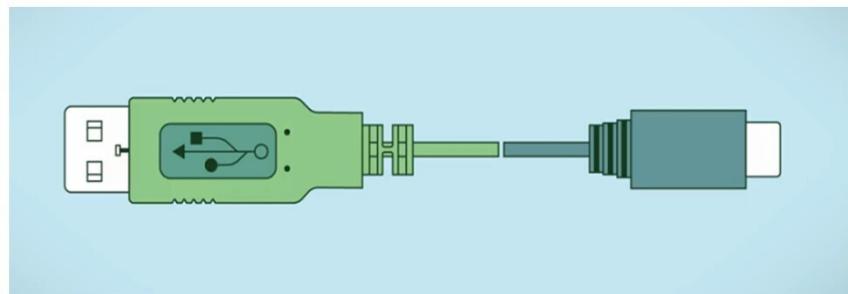
Эти шаблоны сосредоточены на том, как классы наследуют друг от друга и как они составляются из других классов.

И первый шаблон, который мы рассмотрим в этой группе, это Адаптер.

Я уверен, что вы сталкивались с такой ситуацией. У вас есть устройство, которое ожидает один вид разъема, но у вас есть что-то, что имеет другой разъем.

И из-за несоответствия вы не можете установить это соединение.

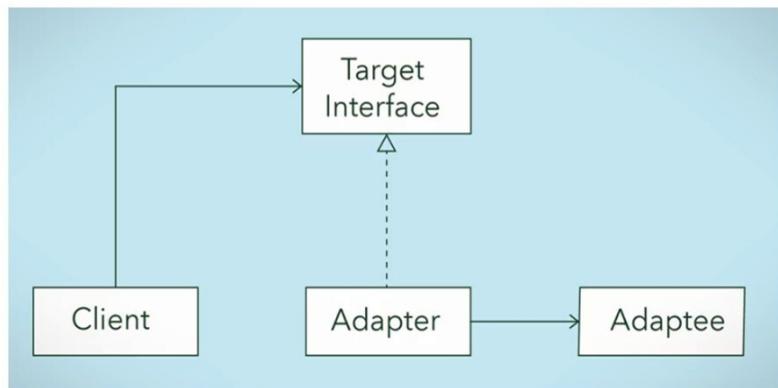
Таким образом, вам нужен адаптер, соединяющий два разъема.



И вы обнаружите, что это является повторяющейся проблемой, когда ваша существующая система должна включать сторонние библиотеки или должна быть подключена к другим системам.

Шаблон проектирования адаптер помогает упростить связь между двумя существующими системами, предоставив совместимый интерфейс.

Этот шаблон дизайна состоит из нескольких частей.



Класс клиента является частью вашей системы, которая хочет использовать стороннюю библиотеку или внешнюю систему.

Adaptee – это класс в сторонней библиотеке или внешней системе, которая будет использоваться.

Класс Adapter находится между клиентом и классом Adaptee.

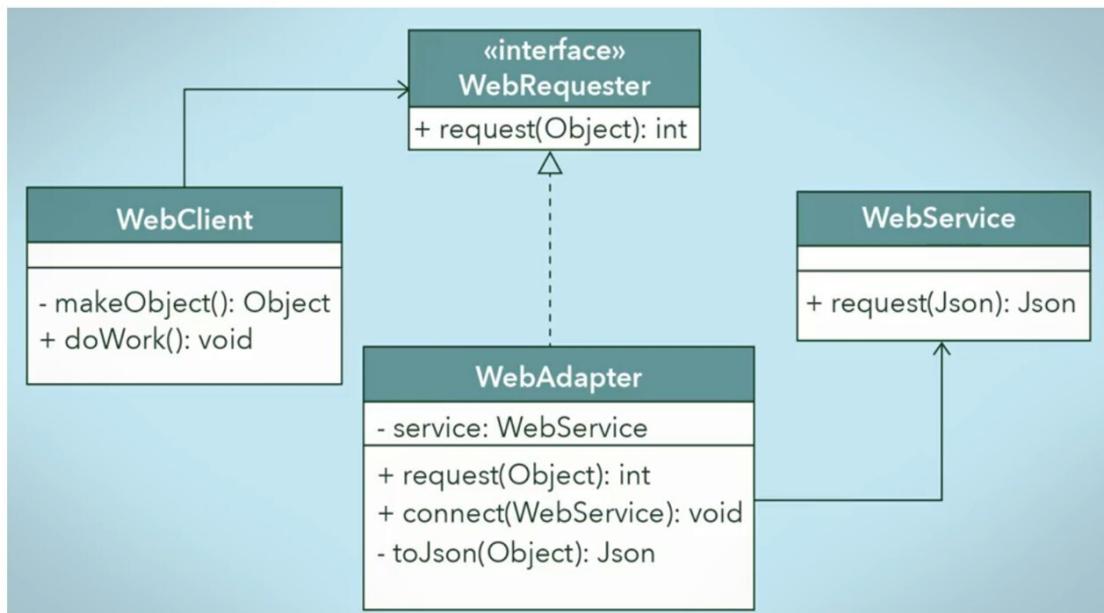
Он реализует целевой интерфейс, который является интерфейсом, который клиент будет использовать.

Адаптер соответствует тому, что ожидает клиент.

Клиент отправляет запрос адаптеру с использованием целевого интерфейса.

Адаптер затем переводит запрос в сообщение, которое будет понимать адаптируемый.

Как только перевод будет завершен, адаптер отправляет переведенный запрос адаптируемому.



В этом примере у нас есть существующий WebClient, который мы хотим использовать для общения с другим WebService.

WebClient может отправить любой объект в запросе, но служба поддерживает только объект Json.

Нам нужно использовать адаптер для преобразования нашего объекта в запросе в объект Json.

Для этого мы проектируем целевой интерфейс.

Мы создаем целевой интерфейс, который будет реализован классом адаптера для класса клиента.

Далее целевой интерфейс реализуется классом адаптера.

И адаптер предоставляет методы, которые будут принимать объект клиента и преобразовывать его в объект Json.

И класс адаптера также выполняет перенос переведенного запроса в адаптируемый сервис.

В вашей основной программе вам необходимо создать экземпляр WebAdapter, WebService и WebClient.

WebClient имеет дело с адаптером через интерфейс WebRequester для отправки запроса.

Обратите внимание, что нашему WebClient не нужно ничего знать о WebService, например, о его потребностях в объектах Json.

Адаптируемый скрыт от клиента классом адаптера.

Теперь, вопрос, почему бы нам просто не изменить интерфейс нашей системы?

Это может сработать. Но что, если другие подсистемы его используют?

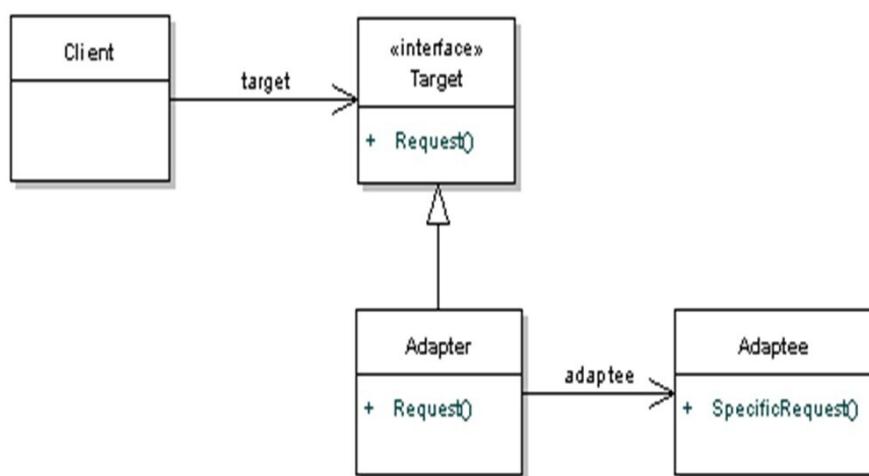
Его изменение может непреднамеренно нарушить другую часть нашей системы.

Не всегда удобно переписывать большие куски вашей системы, чтобы правильно взаимодействовать с новыми сторонними библиотеками или внешними системами.

Просто потому, что ваш интерфейс не соответствует ожидаемому, не означает, что ваша система должна измениться.

Шаблон проектирования адаптера – это метод, который помогает преодолеть разрыв между двумя несовместимыми интерфейсами.

Это позволяет вам продолжать использовать существующие системы и интегрировать в них внешние системы.



Шаблон адаптера работает как мост между двумя несовместимыми интерфейсами.

Этот шаблон определяет один класс, который отвечает за объединение функциональных возможностей независимых или несовместимых интерфейсов.

Примером в реальной жизни может служить кард-ридер, который выступает в качестве адаптера между картой памяти и ноутбуком. Вы вставляете карту памяти в кард-ридер, и кард-ридер вставляете в ноутбук, чтобы можно было считывать карту памяти с помощью ноутбука.

На слайде интерфейс Target определяет интерфейс, используемый клиентом, поэтому клиент взаимодействует с объектами, реализующими интерфейс Target.

С другой стороны, Adaptee – это интерфейс, который нуждается в адаптации, чтобы клиент мог взаимодействовать с ним.

Адаптер адаптирует Adaptee к интерфейсу Target, другими словами, он переводит запрос от клиента к Adaptee адаптируемому.

Рассмотрим применение шаблона адаптера на примере адаптера питания.

В разных странах есть разные типы электрических розеток.

Для того, чтобы разные электрические розетки работали с разными разъемами, необходимо использовать адаптеры.

```
public class GermanElectricalSocket {  
    public void plugIn(GermanPlugConnector plug) {  
        plug.giveElectricity();  
    }  
}  
  
public class UKElectricalSocket {  
    public void plugIn(UKPlugConnector plug) {  
        plug.provideElectricity();  
    }  
}  
  
public interface GermanPlugConnector {  
    public void giveElectricity();  
}  
  
public interface UKPlugConnector {  
    public void provideElectricity();  
}  
  
public class GermanToUKPlugAdapter implements UKPlugConnector {  
    private GermanPlugConnector plug;  
  
    public GermanToUKAdapter(GermanPlugConnector plug) {  
        this.plug = plug;  
    }  
    @Override  
    public void provideElectricity() {  
        plug.giveElectricity();  
    }  
}
```

Здесь у нас есть розетка из Германии, которая принимает только германский разъем.

И есть английская розетка, которая принимает только английский разъем.

Нам нужно сделать адаптер, чтобы английская розетка приняла германский разъем.

Для этого мы создаем класс, который реализует интерфейс английского разъема, чтобы замаскироваться под английский разъем.

Конструктор этого класса принимает германский разъем.

И в методе интерфейса английского разъема, этот класс вызывает метод интерфейса германского разъема.

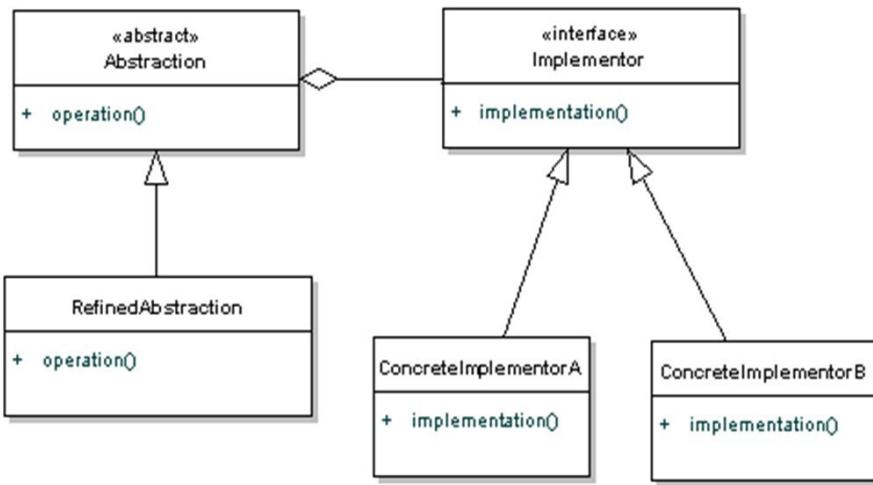
Теперь мы можем передать объект адаптера в метод plugIn английской розетки.

И при вызове метода интерфейса английского разъема, будет на самом деле вызван метод интерфейса германского разъема.

Так работает шаблон адаптера.

Bridge Pattern

Следующий шаблон, который мы рассмотрим, это Мост.



Мост используется, когда нам нужно отделить абстракцию от ее реализации, чтобы они могли разрабатываться независимо.

При использовании наследования реализация жестко привязывается к абстракции, что затрудняет их независимую модификацию.

При коллективной разработке, в больших проектах, иногда бывает необходимо независимо развивать связанные друг с другом абстракцию и реализацию.

При использовании механизма наследования это очень затруднительно.

Любые изменения в абстракции тут же должны быть имплементированы в реализации.

Но мы можем отделить абстракцию от реализации с помощью Моста.

Для этого, на все реализации абстракции заводим общий интерфейс, который эти реализации будут также реализовывать.

В абстракции храним ссылку на интерфейс реализации.

Теперь мы можем совершенно независимо модифицировать абстракцию – путем уточнения абстракции и реализацию – путем реализации интерфейса.

Как правило интерфейс реализации содержит простейшие методы, в то время как абстракция содержит методы более высокого уровня, реализация которых на самом деле является суперпозицией простейших методов из интерфейса реализации.

На первый взгляд, шаблон Мост похож на шаблон Адаптер, в котором класс используется для преобразования одного вида интерфейса в другой.

Однако цель шаблона Адаптер состоит в том, чтобы сделать один или несколько интерфейсов классов похожими на интерфейсы определенного класса.

Шаблон Мост предназначен для отделения интерфейса класса от его реализации, поэтому вы можете изменять или заменять реализацию без изменения кода клиента.

На слайде Abstraction определяет интерфейс абстракции и поддерживает ссылку на объект типа Implementor.

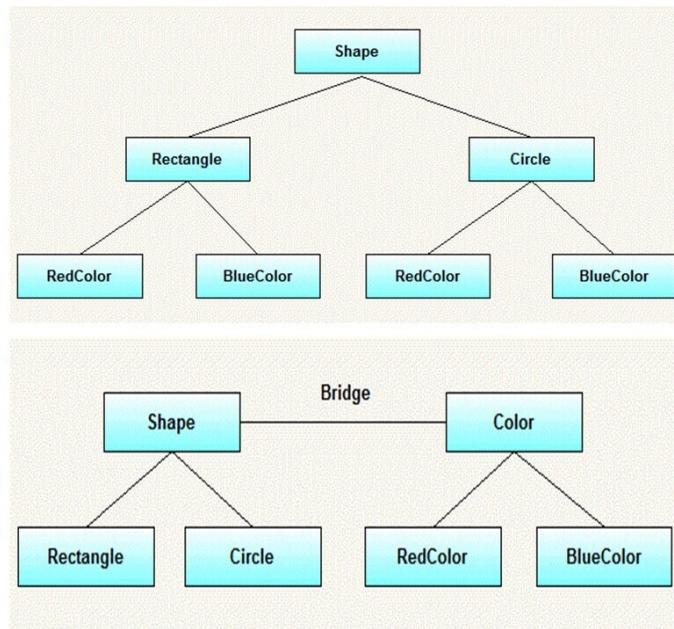
RefinedAbstraction расширяет интерфейс, определяемый абстракцией.

Implementor определяет интерфейс для классов реализации.

Этот интерфейс не должен точно соответствовать интерфейсу абстракции, фактически оба интерфейса могут быть совершенно разными.

Как правило, интерфейс Implementor обеспечивает только примитивные операции, а абстракция определяет операции более высокого уровня, основанные на этих примитивах.

ConcreteImplementor реализует интерфейс Implementor и определяет его конкретную реализацию.



В этом примере интерфейс Shape – это абстракция и она была реализована в виде двух классов RedColor и BlueColor.

Затем мы захотели развить абстракцию и расширили ее как Rectangle и Circle.

Для этого, без шаблона Мост, нам понадобилось перестроить всю иерархию классов – картинка вверху.

С шаблоном Мост, мы можем развивать абстракцию и реализацию независимо – картинка внизу.

Теперь все это в деталях.

```

abstract class Shape {
    Color color;
    Shape(Color color)
    {
        this.color=color;
    }
    abstract public void colorIt();
}

public class Rectangle extends Shape{
    Rectangle(Color color) {
        super(color);
    }

    public void colorIt() {
        System.out.print("Rectangle filled with ");
        color.fillColor();
    }
}

public class Circle extends Shape{
    Circle(Color color) {
        super(color);
    }

    public void colorIt() {
        System.out.print("Circle filled with ");
        color.fillColor();
    }
}

```

У нас есть абстрактный класс `Shape` и его развитие – классы `Rectangle` и `Circle`. Все они содержат ссылку на объект `Color`, который используют при расширении класса `Shape`.

```

public interface Color{
    public void fillColor();
}

public class RedColor implements Color{
    public void fillColor() {
        System.out.println("red color");
    }
}

public class BlueColor implements Color{
    public void fillColor() {
        System.out.println("blue color");
    }
}

public class BridgeDesignPatternMain{
    public static void main(String[] args) {

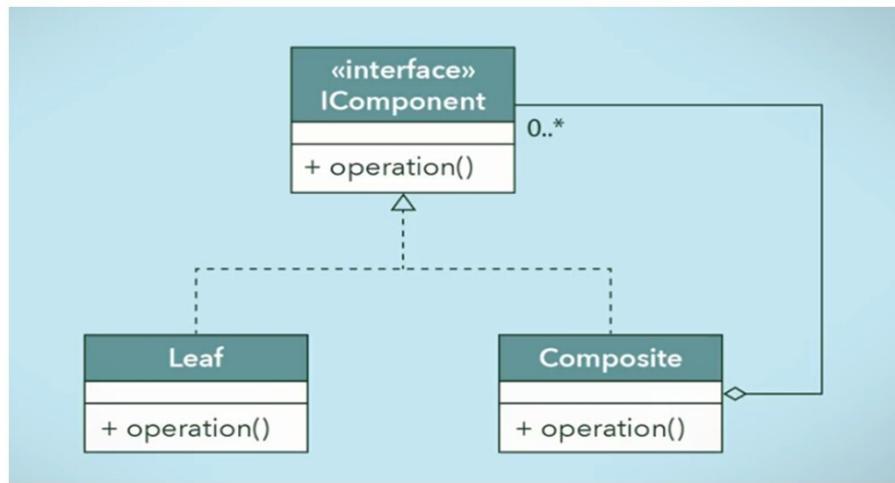
        Shape s1=new Rectangle(new RedColor());
        s1.colorIt();
        Shape s2=new Circle(new BlueColor());
        s2.colorIt();
    }
}

```

И у нас есть интерфейс `Color`, который служит мостом. Этот интерфейс реализуют классы `RedColor` и `BlueColor`. Теперь мы можем создать конкретный объект `Shape`, передав в него конкретный объект `Color`. И эти две иерархии могут развиваться независимо друг от друга.

Composite Pattern

Следующий шаблон, который мы рассмотрим, это Компоновщик.



Компоновщик обеспечивает достижение двух целей.

Это компоновать вложенные структуры объектов и работать с классами этих объектов единообразно.

Интерфейс компонента служит супертипов для набора классов, так что все они могут обрабатываться равномерно.

Это делается путем обеспечения полиморфизма.

Все классы реализации соответствуют одному и тому же интерфейсу.

И вместо интерфейса может использоваться абстрактный суперкласс, так как он также обеспечивать полиморфизм.

Класс компоновщика используется для объединения любых классов, реализующих интерфейс компонента.

Класс компоновщика позволяет вам манипулировать объектами компонентами, которые содержит объект компоновщика.

Класс leaf представляет собой некомпозитный тип.

Он не состоит из других компонентов.

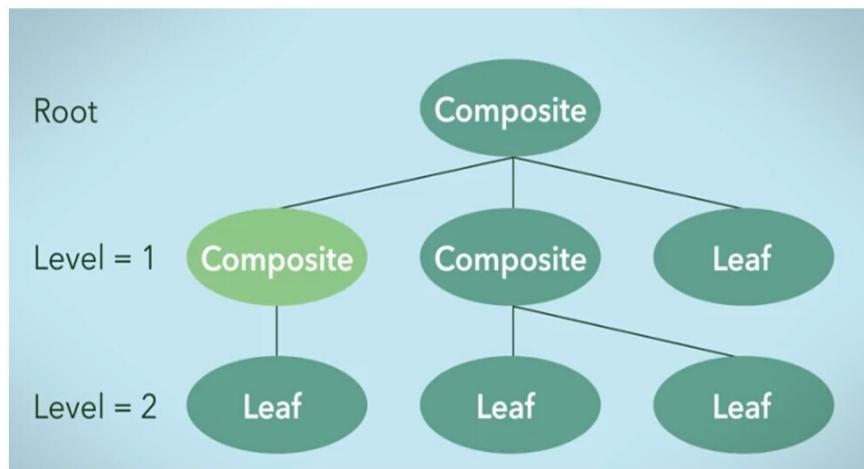
И мы хотим иметь дело с не-композитными и композитными объектами единообразно.

Благодаря тому, что класс листа и класс компоновщика реализуют интерфейс компонента, мы объединяем их в одном типе.

Поэтому классы компоновщика и листа считаются подтипами компонента.

И в зависимости от задачи, вы можете использовать другие классы компоновщика и листа при использовании шаблона проектирования компоновщик.

Но можете использовать только один общий интерфейс компонента или абстрактный суперкласс.



Также объект компоновщика может содержать другие объекты компоновщика, так как класс компоновщика является подтиповом компонента.

Это называется рекурсивной композицией.

Может быть сложно представить, как выглядит этот шаблон дизайна из-за кажущейся циклической природы класса компоновщика.

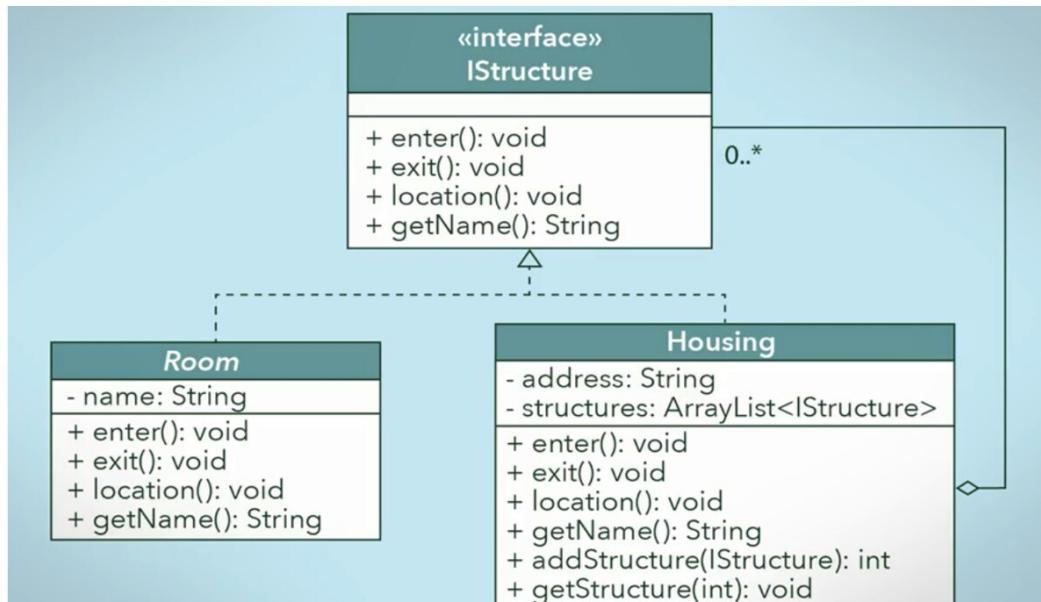
Самый простой способ визуализировать это – думать о нем как о дереве.

На корневом уровне у нас есть наш основной объект компоновщика, который состоит из других объектов компоновщика.

На каждом уровне мы можем добавлять дополнительные объекты компоненты ниже каждого объекта компоновщика.

И обратите внимание, что вы можете добавлять компоненты только к объекту компоновщика, а не к объекту листа.

Это означает, что объект компоновщика имеет потенциал для роста дерева, в то время как объект листа завершает дерево.



Теперь, давайте применим этот шаблон к примеру.

Мы будем использовать пример того, как здания состоят из общих структур.

Эта структура считается интерфейсом компонента.

Он может использоваться для описания здания, этажа или комнаты, несмотря на то, что каждый из них является уникальным типом объектов.

Класс дома представляет собой класс компоновщика.

Дом представляет собой тип структуры, который также может содержать другие структуры.

И комната считается классом листа, так как комната не может содержать другую комнату.

Теперь у нас есть диаграмма классов UML, и давайте реализуем шаблон компоновщика в JAVA.

```

public interface IStructure {
    public void enter();
    public void exit();
    public void location();
    public String getName();
}
  
```

```

public class Housing implements IStructure {
    private ArrayList<IStructure> structures;
    private String address;

    public Housing (String address) {
        this.structures = new ArrayList<IStructure>();
        this.address = address;
    }

    public String getName() {
        return this.address;
    }

    public int addStructure(IStructure component) {
        this.structures.add(component);
        return this.structures.size() - 1;
    }

    public IStructure getStructure(int componentNumber) {
        return this.structures.get(componentNumber);
    }

    public void location() {
        System.out.println("You are currently in " + this.getName() +
            ". It has ");
        for (IStructure struct : this.structures)
            System.out.println(struct.getName());
    }

    /* Print out when you enter and exit the building */
    public void enter() { ... }
    public void exit() { ... }
}
  
```

Во-первых, давайте создадим интерфейс, определяющий общий тип.

Во-вторых, реализуем класс компоновщика.

Мы реализуем интерфейс в этом классе, дав классу дома его собственное поведение, когда его использует клиентский код.

Так как объект дома может состоять из других структур, необходимо представить оболочку с подходящей коллекцией.

Вам также понадобится метод добавления дополнительных компонентов в объект компоновщика.

Здесь мы использовали ArrayList и метод addStructure для добавления в него компонентов.

```
public abstract class Room implements IStructure {  
    public String name;  
  
    public void enter() {  
        System.out.println("You have entered the " + this.name);  
    }  
  
    public void exit() {  
        System.out.println("You have left the " + this.name);  
    }  
  
    public void location() {  
        System.out.println("You are currently in the " + this.name);  
    }  
  
    public String getName() {  
        return this.name;  
    }  
}
```

И третий шаг, реализуем класс листа.

Так как класс leaf не содержит каких-либо компонентов, нам не нужно добавлять в него набор компонентов или любые методы управления коллекцией компонентов.

Нам просто нужно реализовать методы интерфейса компонента.

Теперь, когда мы закончили реализацию здания с использованием шаблона проектирования компоновщик, как мы его фактически используем?

```

public class Program {

    public static void main(String args[]) {
        Housing building = new Housing("123 Street");
        Housing floor1 = new Housing("123 Street - First Floor");
        int firstFloor = building.addStructure(floor1);

        Room washroom1m = new Room("1F Men's Washroom");
        Room washroom1w = new Room("1F Women's Washroom");
        Room common1 = new Room("1F Common Area");

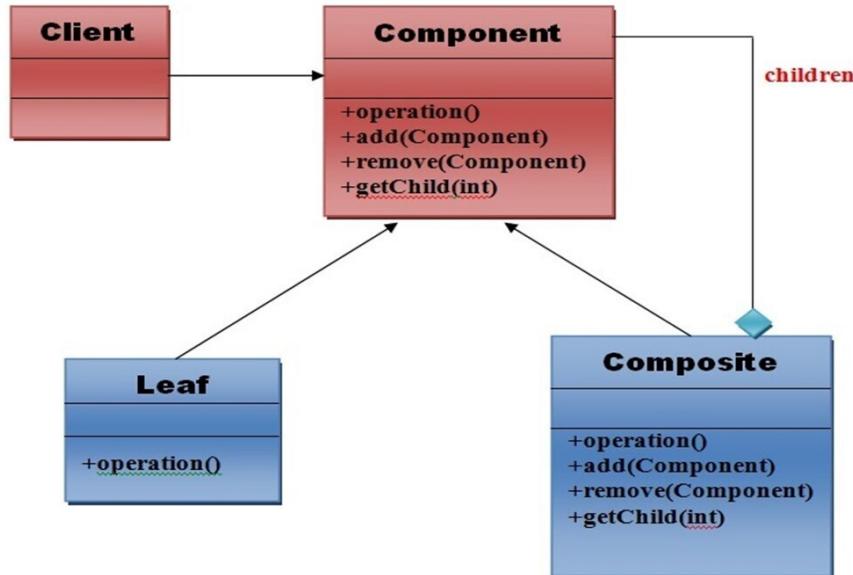
        int firstMens = floor1.addStructure(washroom1m);
        int firstWomans = floor1.addStructure(washroom1w);
        int firstCommon = floor1.addStructure(common1);

        building.enter(); // Enter the building
        Housing currentFloor = building.getStructure(firstFloor);
        currentFloor.enter(); // Walk into the first floor
        Room currentRoom = currentFloor.getStructure(firstMens);
        currentRoom.enter(); // Walk into the men's room
        currentRoom = currentFloor.getStructure(firstCommon);
        currentRoom.enter(); // Walk into the common area
    }
}

```

Мы просто строим здание по структуре.

В этом примере наше здание имеет один этаж, в котором есть общая комната и две ванные комнаты.



Таким образом, шаблон Компоновщик используется там, где нужно обрабатывать группу объектов аналогично одному объекту.

Этот шаблон создает класс, содержащий группу собственных объектов.

И этот класс предоставляет способы изменения группы одинаковых объектов.

Здесь клиент использует интерфейс класса Компонент для взаимодействия с объектами в структуре композиции.

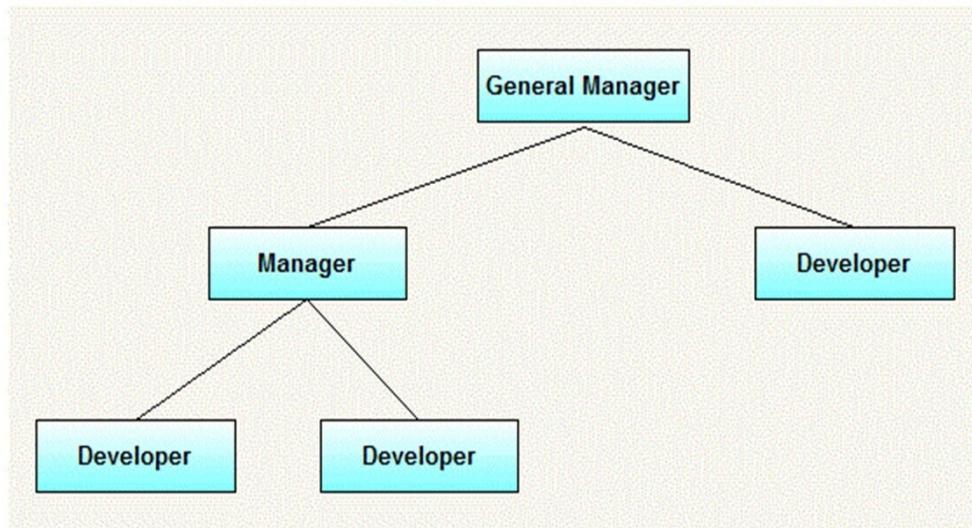
Шаблон Компоновщик обрабатывает каждый узел структуры двумя способами: Composite или leaf.

Composite означает, что он может иметь другие объекты под ним.

Лист означает, что у него нет объектов под ним.

Если получателем запроса является лист, запрос будет обрабатываться напрямую.

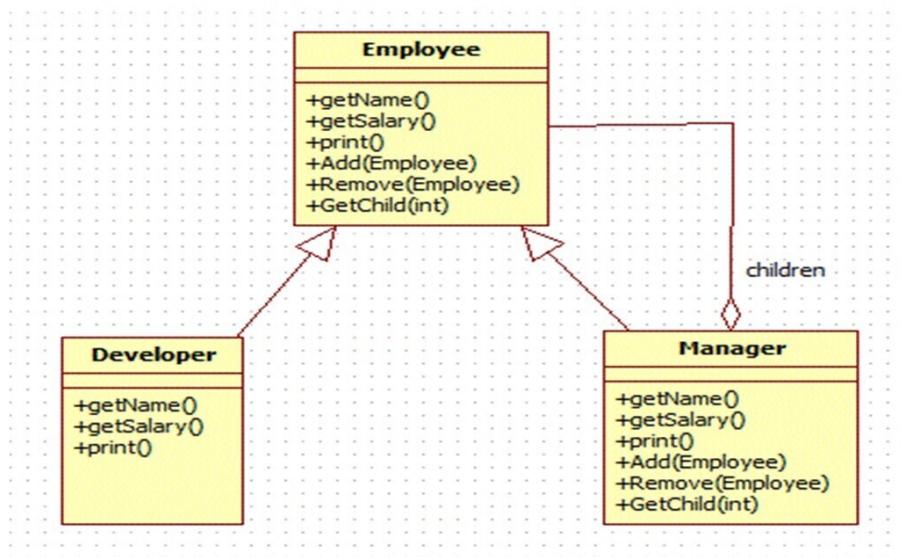
Если получателем является Composite, то он обычно отправляет запрос своему ребенку для выполнения дополнительных операций.



В этом примере на верхней позиции есть генеральный менеджер.

И у него в подчинении есть два сотрудника, один из них – менеджер, а другой – разработчик, и у менеджера в подчинении есть два разработчика.

Мы хотим распечатать имя и заработную плату всех сотрудников сверху донизу.



Сначала мы создадим интерфейс Компонент, который представляет объект в композиции.

Этот интерфейс содержит все общие операции, которые применимы как к менеджеру, так и к разработчику.

```
public class Manager implements Employee{  
    private String name;  
    private double salary;  
    private List employees = new ArrayList();  
    public Manager(String name,double salary){  
        this.name = name;  
        this.salary = salary;  
    }  
    public void add(Employee employee) {  
        employees.add(employee);  
    }  
    public String getName() {  
        return name;  
    }  
    public double getSalary() {  
        return salary;  
    }  
    public void print() {  
        Iterator employeeIterator = employees.iterator();  
        while(employeeIterator.hasNext()) {  
            Employee employee = employeeIterator.next();  
            System.out.println("Name ="+ employee.getName());  
            System.out.println("Salary =" + employee.getSalary());  
        }  
    }  
    public void remove(Employee employee) {  
        employees.remove(employee);  
    }  
}  
  
public interface Employee {  
    public void add(Employee employee);  
    public void remove(Employee employee);  
    public String getName();  
    public double getSalary();  
    public void print();  
}
```

Далее мы создадим менеджера.

Ключевым моментом здесь является то, что все общие методы делегируют свои операции дочерним объектам.

У этого класса есть методы доступа и изменения его дочерних элементов.

```
public class Developer implements Employee{  
  
    private String name;  
    private double salary;  
  
    public Developer(String name,double salary){  
        this.name = name;  
        this.salary = salary;  
    }  
    public void add(Employee employee) {  
    }  
    public String getName() {  
        return name;  
    }  
    public double getSalary() {  
        return salary;  
    }  
    public void print() {  
        System.out.println("Name =" +getName());  
        System.out.println("Salary =" +getSalary());  
    }  
    public void remove(Employee employee) {  
    }  
}
```



```
public interface Employee {  
    public void add(Employee employee);  
    public void remove(Employee employee);  
    public String getName();  
    public double getSalary();  
    public void print();  
}
```

Дале мы создадим класс разработчика.

Этот класс является листовым узлом, поэтому все операции, связанные с доступом к дочерним элементам, будут пустыми, так как у него нет детей.

```
Employee emp1=new Developer("John", 10000);
Employee emp2=new Developer("David", 15000);
Employee manager1=new Manager("Daniel",25000);
manager1.add(emp1);
manager1.add(emp2);
Employee emp3=new Developer("Michael", 20000);
Manager generalManager=new Manager("Mark", 50000);
generalManager.add(emp3);
generalManager.add(manager1);
generalManager.print();
```

Теперь мы можем через интерфейс Employee, который является интерфейсом компонента, добавлять менеджеров и разработчиков в единой манере и применять к ним общие операции.

Decorator Pattern

Следующий шаблон, который мы рассмотрим, это Декоратор или Обертка. Кофе нравится многим людям.



Некоторым нравится простое черное кофе, а некоторые предпочитают латте или другие сложные версии кофе.

Добавление ингредиентов, таких как молоко, ароматизированный крем или горячий шоколад, – это способы изменения кофе.

Мы все еще можем думать о нем как о напитке на основе кофе, но есть ожидаемые различия между черным кофе и более сложными вариантами кофе, такими как латте или эспрессо.

Независимо от того, какой тип кофе мы пьем, эффекты каждого используемого ингредиента не меняются.

Например, кофе всегда будет действовать как стимулятор из-за содержания в нем кофеина.

Это не изменится, если мы добавим молоко, кофе по-прежнему будет стимулятором. И это справедливо для всего, что вы добавляете в кофе.

Несмотря на то, что каждый ингредиент продолжает сохранять свои собственные характеристики, объединение их создает новый напиток, который может быть более приятным, чем при употреблении ингредиентов отдельно.

В программном обеспечении, полезно иметь гибкие комбинации общего поведения.

Но мы сталкиваемся с проблемой, пытаясь сделать это динамически во время выполнения, потому что поведение объекта определяется его классом, но понятие класса и отношений, подобных наследованию, являются статическими.

То есть, определяются во время компиляции.

Это означает, что мы не можем вносить изменения в классы во время работы нашей программы.

Нам нужно создать новый класс для получения новой комбинации поведения.

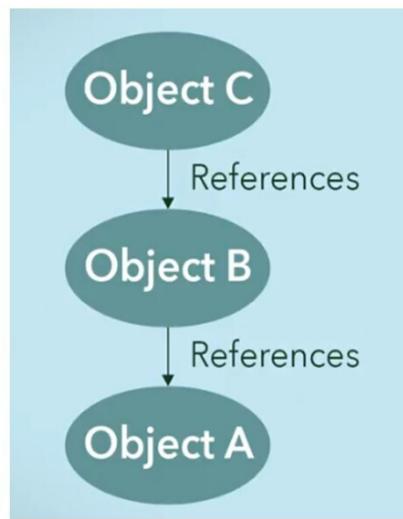
В результате, наличие множества новых комбинаций приведет к большому количеству классов, и мы этого не хотим.

Имея объект с определенным поведением, можем ли мы динамически добавлять к нему дополнительные поведения или обязанности?

К счастью, для этого есть шаблон декоратора, который использует агрегацию для комбинирования поведения во время выполнения.

Агрегация используется для представления отношения "has a" «имеет» между двумя объектами.

Мы можем использовать это отношение "has a" «имеет» для построения стека объектов, где каждый уровень стека содержит объект, который знает о своем собственном поведении и дополняет то поведение, которое находится под ним в стеке.



Вот как выглядит стек агрегации.

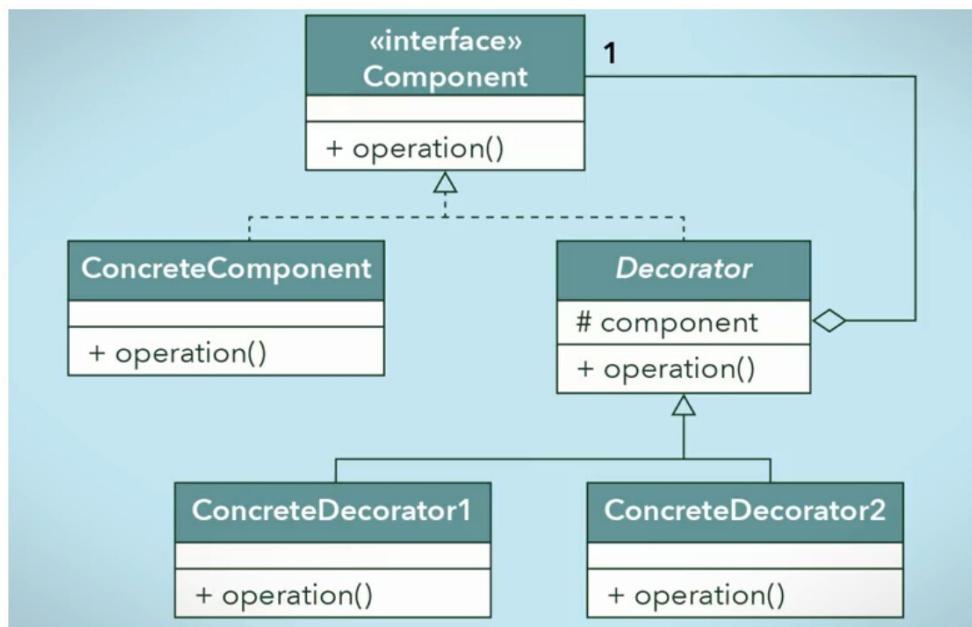
Объект А является базовым объектом, так как он не содержит другого объекта.

Он имеет свой собственный набор поведений.

Объект В агрегирует объект А, позволяя объекту В, по сути, дополнять поведение объекта А.

Мы можем продолжать добавлять объекты в стек таким образом, чтобы объекты агрегировали низлежащие объекты, дополняя их поведение.

Соотношение агрегирования всегда взаимно однозначно в шаблоне проектирования декоратор, чтобы создать стек таким образом, чтобы один объект находился поверх другого.



Чтобы достичь общей комбинации поведения для объектов в стеке, вы должны вызвать верхний объект.

Этот объект будет вызывать низлежащий объект и так далее.

Самый нижний объект ответит своим поведением, затем выше лежащие объекты будут добавлять свое поведение.

Фактическая структура этого шаблона проектирования использует как интерфейсы, так и наследование, так что классы соответствуют общему типу, экземпляры которых могут быть в стеке в совместимом виде для создания согласованной комбинации поведения в целом.

Здесь интерфейс компонента используется для определения общего типа для всех классов.

Класс клиента ожидает такой же интерфейс для всех классов компонентов.

Конкретный класс компонентов реализует интерфейс компонента.

Экземпляр этого класса можно использовать в качестве базового объекта в стеке.

Декоратор – это абстрактный класс.

И подобно конкретному классу компонента, он реализует интерфейс компонента.

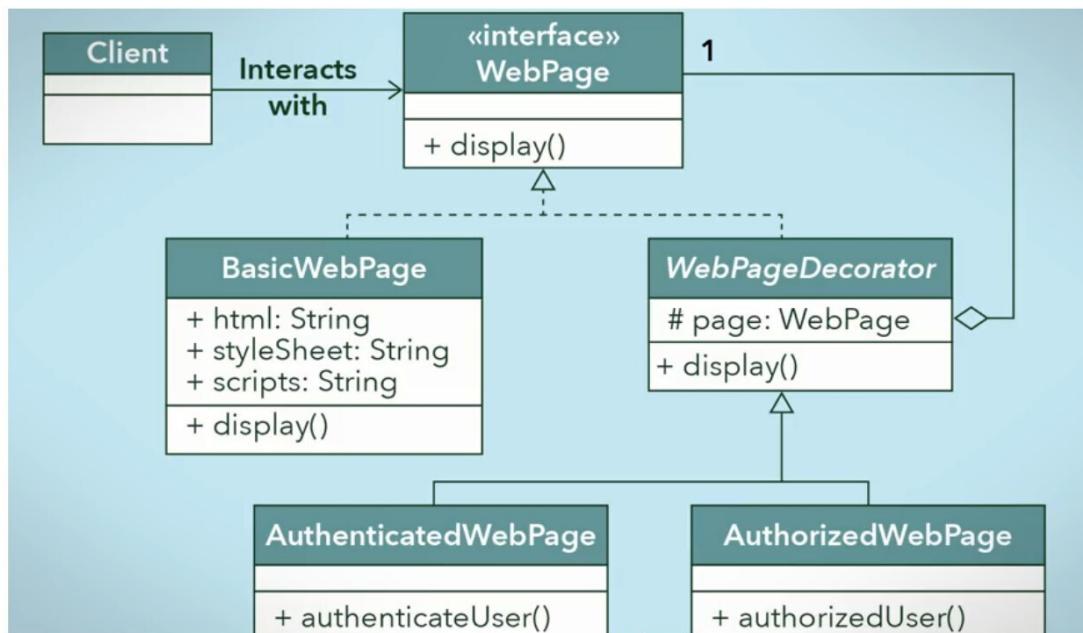
Основное отличие заключается в том, что декоратор агрегирует другие типы компонентов, позволяя нам складывать компоненты друг над другом в стеке, и декоратор служит абстрактным суперклассом классов, которые обеспечивают прирост поведения.

Мы создаем стек компонентов, начиная с экземпляра конкретного класса компонента и продолжая экземплярами подклассов абстрактного класса декоратора.

Что касается нашей аналогии с кофе, конкретным компонентом будет черный кофе.

Декораторами для нашего кофе будут молоко, сахар, горячий шоколад и любые другие ингредиенты, которые вы можете добавить в кофе.

Давайте рассмотрим пример, иллюстрирующий применение шаблона декоратора.



Рассмотрим пример с веб-страницей.

Базовая веб-страница – это просто разметка с использованием HTML со списками стилей и, возможно, с Javascript.

Однако поведение веб-страницы может быть более сложным.

Что делать, если вы хотите, чтобы пользователи, обращающиеся к вашей странице, авторизовались?

Или если вы хотите разбить большое количество результатов поиска на отдельные страницы?

Вы не хотите писать совершенно разные типы веб-страниц для каждой возможной комбинации разрешений доступа к веб-странице, разбиения ее на страницы или кеширования.

Вместо этого вы можете использовать шаблон декоратора, чтобы создать один класс для каждого типа поведения и создать определенную комбинацию веб-страниц, которая нужна вам во время выполнения.

Здесь интерфейс компонента – это веб-страница.

Он определяет все подклассы в шаблоне как тип веб-страница, которые имеют собственную реализацию того, как отображаться.

Конкретный класс компонента – это ваша базовая веб-страница.

Эта базовая веб-страница должна знать, как отображать все элементы веб-страницы.

Теперь нам нужны декораторы, чтобы добавить больше функциональности в базовую веб-страницу, используя агрегацию вместо создания подклассов базовой веб-страницы.

Для этого вам нужно использовать подтипы абстрактного класса декоратора веб-страницы, чтобы расширить базовую веб-страницу.

И декоратор веб-страницы будет подтипом типа веб-страницы.

Поэтому подтип декоратора также является подтипом интерфейса веб-страницы.

Вы можете определить любое количество дополнительных действий, которыми вы хотите дополнить базовую веб-страницу.

В этом примере мы улучшаем базовую веб-страницу, добавляя авторизацию, чтобы обеспечить пользователю доступ к странице и аутентификацию, чтобы убедиться, что пользователь является тем, кем он себя называет.

Как вы видите, это уменьшает количество классов, которые нам нужно создать.

Если бы вы использовали наследование базовой веб-страницы, вам нужно было бы создать класс для каждой комбинации этих поведений.

Это означает, что для совместной функции авторизации и аутентификации нам потребуется еще один отдельный класс.

Модель декоратора решает эту проблему, позволяя конкретным декораторам агрегировать компоненты.

```
public interface WebPage {  
    public void display();  
}  
  
public class BasicWebPage implements WebPage {  
    private String html = ...;  
    private String styleSheet = ...;  
    private String scripts = ...;  
  
    public void display() {  
        /* Renders the HTML to the stylesheet, and run any  
         * embedded scripts */  
        System.out.println("Basic web page");  
    }  
}
```

Теперь давайте посмотрим, как реализовать этот проект.

Во-первых, спроектируйте интерфейс компонента.

Вы определяете интерфейс для остальных классов в шаблоне проектирования, которые будут подтипами этого типа.

Интерфейс определит общее поведение, которое будет иметь ваша базовая веб-страница и декораторы.

Во-вторых, реализуем интерфейс с базовым классом конкретного компонента.

Ваша базовая веб-страница будет реализовывать то, как она будет отображаться, используя стандартную HTML-разметку и стили.

На вашей базовой веб-странице также будет запущен базовый Javascript код.

Это будет базовый строительный блок для всех объектов веб-страниц во время выполнения.

```
public abstract class WebPageDecorator implements WebPage {  
    protected WebPage page;  
  
    public WebPageDecorator(WebPage webpage) {  
        this.page = webpage;  
    }  
  
    public void display() {  
        this.page.display();  
    }  
}
```

Шаг третий, реализуем интерфейс с абстрактным классом декоратора.

Реализация класса декоратора важна, несмотря на то, что у него мало кода.

Первое, что нужно отметить, это то, что декоратор веб-страниц содержит только один экземпляр веб-страницы.

Это позволяет нам складывать декораторы поверх основной веб-страницы и друг над другом в стеке.

Каждый тип веб-страницы несет ответственность за свое поведение и будет рекурсивно ссылаться на следующую веб-страницу в стеке, чтобы выполнить ее поведение.

И конструктор позволит связать разные подтипы веб-страницы вместе в стеке.

```
public class AuthorizedWebPage extends WebPageDecorator {  
    public AuthorizedWebPage(WebPage decoratedPage) {  
        super(decoratedPage);  
    }  
    public void authorizedUser() {  
        System.out.println("Authorizing user");  
    }  
    public display() {  
        super.display();  
        this.authorizedUser();  
    }  
}  
  
public class AuthenticatedWebPage extends WebPageDecorator {  
    public AuthenticatedWebPage(WebPage decoratedPage) {  
        super(decoratedPage);  
    }  
    public void authenticateUser() {  
        System.out.println("Authenticating user");  
    }  
    public display() {  
        super.display();  
        this.authenticateUser();  
    }  
}
```

И шаг 4, мы наследуем от абстрактного декоратора и реализуем интерфейс компонента с классами конкретных декораторов.

Конструкторы будут использовать конструктор абстрактного суперкласса, так как это позволит нам складывать декораторы вместе в стек.

Помните, что абстрактный класс декоратора обрабатывает агрегацию классов конкретных декораторов.

У каждого декоратора есть свои обязанности.

И вы реализуете эти обязанности в соответствующих классах, чтобы их можно было вызвать.

Чтобы рекурсивно вызвать поведение, декораторы вызывают метод суперкласса в своем методе `display`.

Так как абстрактный суперкласс декоратора обеспечивает агрегацию различных типов веб-страниц, вызов `super.display` приведет к следующей веб-странице в стеке, чтобы выполнить его версию метода `display`, до тех пор, пока вы не перейдете на основную веб-страницу.

Идея здесь состоит в том, чтобы связать вызовы вплоть до нижнего объекта, и отправить их выполнение назад.

Это имеет смысл, потому что вам нужно создать базовую веб-страницу, прежде чем вы сможете добавить к ней больше поведений.

```
public class Program {  
    public static void main(String args[]) {  
        WebPage myPage = new BasicWebPage();  
        myPage = new AuthorizedWebPage(myPage);  
        myPage = new AuthenticatedWebPage(myPage);  
        myPage.display();  
    }  
}
```

Теперь, давайте посмотрим все это в действии.

Сначала вы создаете базовую веб-страницу, затем добавляете поведение авторизации, а затем добавляете поведение аутентификации.

Таким образом, когда вызывается метод `display`, он связывает вызовы метода с базовой веб-страницей.

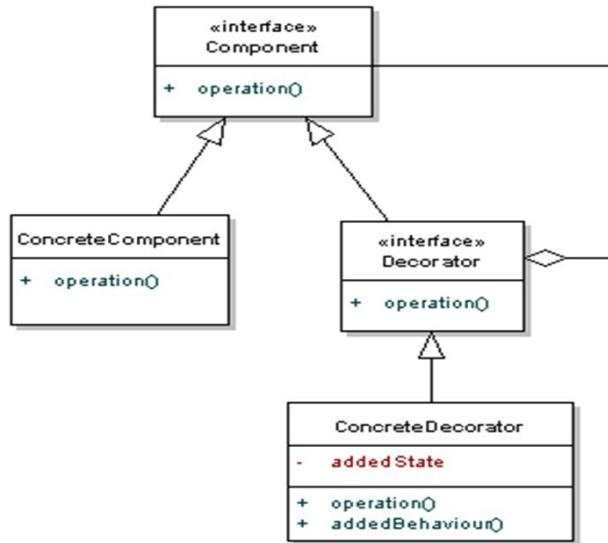
Таким образом, этот шаблон позволяет динамически наращивать поведение базовой веб-страницы, так как мы можем создавать и добавлять новые объекты в стек, используя агрегацию вместо чистого наследования.

Полиморфизм достигается за счет реализации единого интерфейса.

И агрегация позволяет нам создавать стек объектов.

И, объединив агрегацию и полиморфизм, мы можем рекурсивно вызывать одно и то же поведение по стеку, и поведение должно выполняться вверх обратно.

Шаблон декоратора не только позволяет динамически изменять объекты, но также уменьшает разнообразие классов, нужно написать.



Таким образом, шаблон Decorator позволяет пользователю добавлять новые функции к существующему объекту без изменения его структуры.

Этот шаблон создает класс Декоратор, который обертывает исходный класс и предоставляет дополнительную функциональность, сохраняя сигнатуру методов класса нетронутой.

Концепция декоратора заключается в том, что он добавляет дополнительные атрибуты к объекту динамически.

Другими словами, шаблон Decorator использует композицию вместо наследования для расширения функциональности объекта во время выполнения.

Здесь на слайде, Компонент определяет интерфейс для объектов, которые могут иметь динамические изменения, а ConcreteComponent – это просто реализация этого интерфейса.

Декоратор имеет ссылку на Компонент, а также соответствует интерфейсу Component.

Это важно помнить, поскольку Decorator по сути обертывает Компонент.

ConcreteDecorator просто добавляет функциональность Компоненту.

```

public interface Car {
    public void assemble();
}

public class BasicCar implements Car {
    @Override
    public void assemble() {
        System.out.print("Basic Car.");
    }
}

public class SportsCar extends CarDecorator {
    public SportsCar(Car c) {
        super(c);
    }

    @Override
    public void assemble(){
        super.assemble();
        System.out.print(" Adding features of Sports Car.");
    }
}

```

```

public class CarDecorator implements Car {
    protected Car car;
    public CarDecorator(Car c){
        this.car=c;
    }

    @Override
    public void assemble() {
        this.car.assemble();
    }
}

Car sportsCar = new SportsCar(new BasicCar());
sportsCar.assemble();
System.out.println("\n*****");

```

В этом примере интерфейс Car – это интерфейс Компонент.

Класс BasicCar – это реализация интерфейса Компонент.

Класс Decorator реализует интерфейс Компонент.

И переменная компонента должна быть доступна для дочерних классов декоратора, поэтому мы делаем эту переменную `protected`.

Класс SportsCar расширяет базовую функциональность декоратора и соответственно изменяет поведение компонента.

То есть в классе SportsCar, мы передаем в конструктор базовый класс Car и устанавливаем его как поле класса.

Затем мы вызываем декорированный метод `assemble`, в котором сначала вызывается метод `assemble` базового класса Car, а затем добавляется дополнительное поведение.

Facade Pattern

Следующий шаблон, который мы рассмотрим, это Фасад.

По мере того как части вашей системы становятся больше, они, естественно, становятся более сложными.

Однако сложность системы не всегда является признаком плохого дизайна.

Объем задачи, которую вы пытаетесь решить, может быть настолько большим, что требует сложного решения.

Однако классы клиентов предпочли бы более простое взаимодействие.

И шаблон проектирования фасад обеспечивает единый упрощенный интерфейс для клиентских классов для взаимодействия с подсистемой.

Фасад - это класс-оболочка, который инкапсулирует подсистему, чтобы скрыть сложность подсистемы

Что такое фасад?

Если вы когда-либо ходили в магазин, или в ресторане или заказывали что-либо в Интернете, у вас был опыт взаимодействия с шаблоном фасада в реальном мире.

Представьте себе, что вы идете по улице, ища место, где можно поужинать.

Что вы ищете, когда находитесь на улице?

Естественно, вы ищете знак на фасаде здания, который указывает, что его содержимое может предоставить вам сервис кафе или ресторана.

Точно так же, когда вы совершаете покупки в Интернете, вы смотрите на внешний указатель, что на веб-сайте есть виртуальный магазин.

Эти индикаторы, направленные наружу, используются для информирования о том, какие виды услуг доступны.

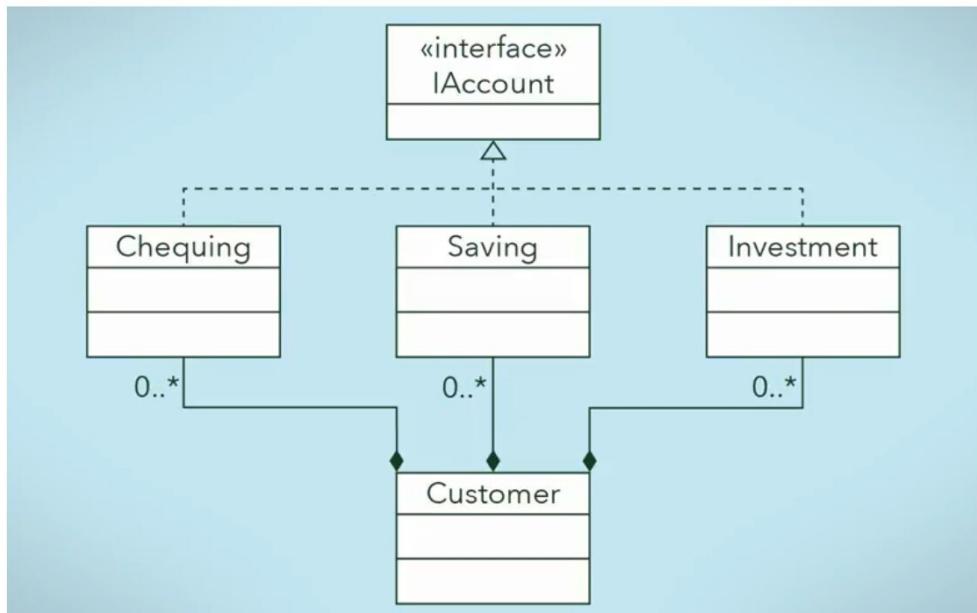
Вы можете приобретать товары и услуги, не зная, как обрабатывается запрос.

За фасадами находятся реализации этих услуг.

Имейте в виду, что фасад фактически не добавляет больше функциональности, фасад просто выступает в качестве точки входа в подсистему.

В программном обеспечении фасад – это класс-оболочка, который инкапсулирует подсистему, чтобы скрыть сложность подсистемы.

Этот класс-оболочка позволяет классу клиента взаимодействовать с подсистемой через фасад.

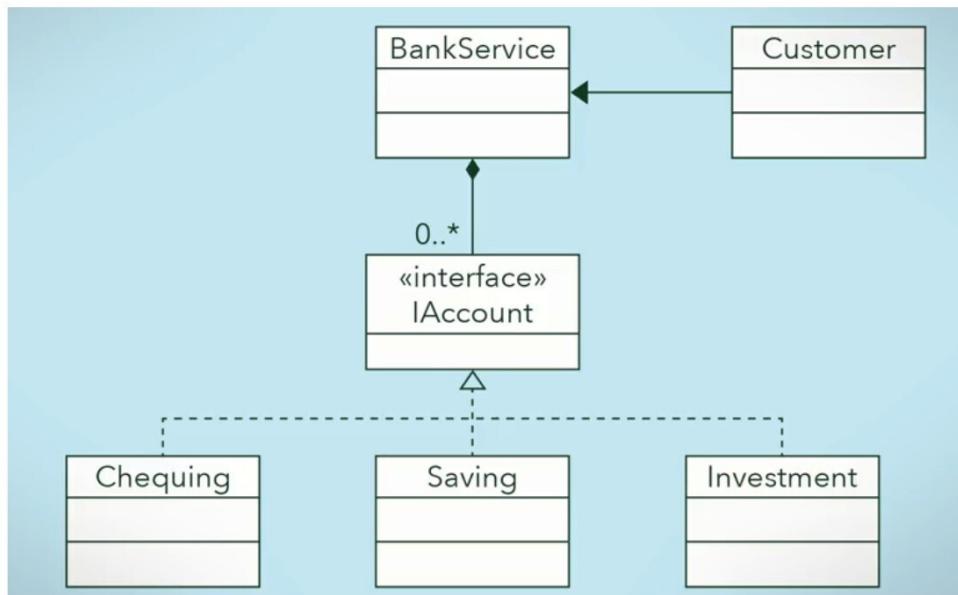


Давайте посмотрим, как клиентский код будет взаимодействовать с подсистемой без класса фасада для простой банковской системы.

Без класса фасада класс клиента будет содержать экземпляры классов текущего счета, сберегательного и инвестиционного счетов.

Это означает, что клиент несет ответственность за правильное создание каждого из этих классов и знает обо всех их различных атрибутах и методах.

Это похоже на управление всеми вашими финансовыми счетами в реальной жизни, что может быть очень сложным при большом количестве счетов, вместо того чтобы позволить финансовому учреждению делать это за вас.



Вместо этого мы представляем класс банковских услуг, который действует как фасад для классов текущего, сберегательного и инвестиционного счетов.

Клиенту больше не нужно обрабатывать экземпляры счетов или решать какие-либо другие сложности финансового управления.

Поскольку три разных счета реализуют интерфейс IAccount, класс банковского обслуживания эффективно обертывает классы реализации интерфейса и предоставляет более простой интерфейс для использования классом клиента.

```
public interface IAccount {  
    public void deposit(BigDecimal amount);  
    public void withdraw(BigDecimal amount);  
    public void transfer(BigDecimal amount);  
    public int getAccountNumber();  
}
```

```
public class Chequing implements IAccount { ... }  
public class Saving implements IAccount { ... }  
public class Investment implements IAccount { ... }
```

Для реализации этого шаблона, во-первых, создайте интерфейс.

Этот Java-интерфейс будет реализован различными классами счетов и не будет известен классу клиентов.

Во-вторых, реализуйте интерфейс одним или несколькими классами.

Помните, что интерфейсы позволяют создавать подтипы, что означает, что текущий, сберегательный и инвестиционный счета являются подтипами i-Account и, как ожидается, будут вести себя как тип счета.

В этом примере вы только реализуете и скрываете один интерфейс, но на практике класс фасада может использоваться для обертывания всех интерфейсов и классов подсистемы.

```
public class BankService {  
    private Hashtable<int, IAccount> bankAccounts;  
    public BankService() {  
        this.bankAccounts = new Hashtable<int, IAccount>;  
    }  
    public int createNewAccount(String type, BigDecimal initAmount) {  
        IAccount newAccount = null;  
        switch (type) {  
            case "chequing":  
                newAccount = new Chequing(initAmount);  
                break;  
            case "saving":  
                newAccount = new Saving(initAmount);  
                break;  
            case "investment":  
                newAccount = new Investment(initAmount);  
                break;  
            default:  
                System.out.println("Invalid account type");  
                break;  
        }  
        if (newAccount != null) {  
            this.bankAccounts.put(newAccount.getAccountNumber(), newAccount);  
            return newAccount.getAccountNumber();  
        }  
        return -1;  
    }  
    public void transferMoney(int to, int from, BigDecimal amount) {  
        IAccount toAccount = this.bankAccounts.get(to);  
        IAccount fromAccount = this.bankAccounts.get(from);  
        fromAccount.transfer(toAccount, amount);  
    }  
}
```

И третий шаг.

Создайте класс фасада и оберните классы, реализующие интерфейс.

Класс банковского обслуживания – это фасад.

Как он это делает?

Обратите внимание, что его публичные методы просты в использовании и не показывают базовый интерфейс и реализующие его классы.

Еще одно замечание заключается в том, что мы устанавливаем модификаторы доступа для каждого счета как приватные.

Так как цель шаблона фасада заключается в том, чтобы скрыть сложность, мы используем принцип скрытия информации, чтобы классы клиентов не видели объекты счетов и как ведут себя эти счета.

```
public class Customer {  
  
    public static void main(String args[]) {  
        BankService myBankService = new BankService();  
  
        int mySaving = myBankService.createNewAccount("saving",  
            new BigDecimal(500.00));  
  
        int myInvestment = myBankService.createNewAccount(  
            "investment", new BigDecimal(1000.00));  
  
        myBankService.transferMoney(mySaving, myInvestment, new  
            BigDecimal(300.00));  
    }  
}
```

И четвертый шаг.

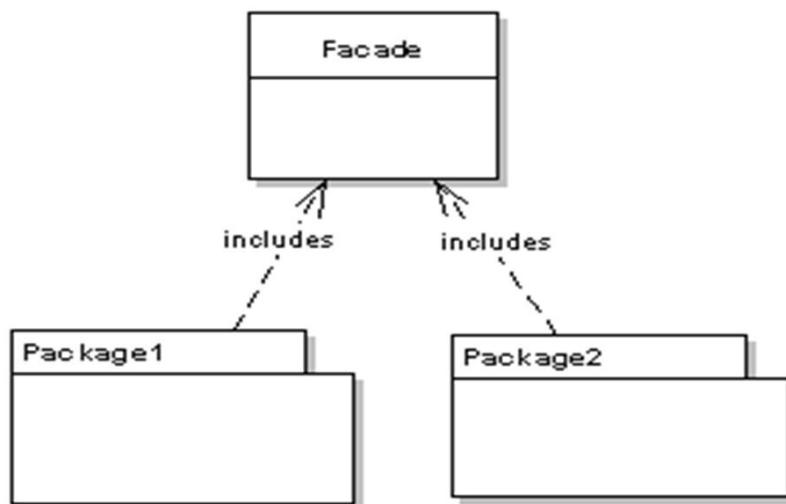
Используйте класс фасада для доступа к подсистеме.

Классы клиентов могут обращаться к функциям разных счетов с помощью методов класса BankService.

Класс BankService показывает клиенту, какие действия он разрешает клиенту, а затем передает это действие соответствующим объектам счетов.

Теперь, когда у нас есть фасад, клиентский класс может получить доступ к своим счетам через класс BankService.

Мы эффективно скрыли сложность управления счетами от клиента, используя класс фасадов BankService.



Таким образом, шаблон Фасад скрывает сложность системы и обеспечивает интерфейс для клиента, с помощью которого клиент может получить доступ к системе.

Этот шаблон включает в себя один класс, который обеспечивает упрощенные методы, требуемые клиентом, и делегирует вызовы методам существующих системных классов.

Шаблон Фасад обеспечивает унифицированный и упрощенный интерфейс к набору интерфейсов в подсистеме, поэтому он скрывает сложности подсистемы от клиента.

Другими словами, шаблон Фасад описывает интерфейс более высокого уровня, который упрощает использование подсистемы.

Таким образом, каждая абстрактная фабрика является фасадом.

Мы можем легко найти фасады в реальном мире.

Операционные системы – один из таких примеров – вы не видите всех внутренних деталей вашего компьютера, но ОС предоставляет упрощенный интерфейс для использования машины.

Также, как и шаблон адаптера, фасад можно использовать для скрытия внутренней работы сторонней библиотеки или некоторого кода.

Все, что нужно делать клиенту, – это взаимодействовать с фасадом, а не с подсистемой, которую он охватывает.

```

public class MySqlHelper{
    public static Connection getMySqlDBConnection(){
        //get MySql DB connection using connection parameters
        return null;
    }
    public void generateMySqlPDFReport(String tableName, Connection con){
        //get data from table and generate pdf report
    }
    public void generateMySqlHTMLReport(String tableName, Connection con){
        //get data from table and generate pdf report
    }
}
public class OracleHelper {
    public static Connection getOracleDBConnection(){
        //get Oracle DB connection using connection parameters
        return null;
    }
    public void generateOraclePDFReport(String tableName, Connection con){
        //get data from table and generate pdf report
    }
    public void generateOracleHTMLReport(String tableName, Connection con){
        //get data from table and generate pdf report
    }
}

```

Предположим, что у нас есть два класса, которые помогают работать с двумя разными базами данных.

Клиентскому коду нужно выбирать, с каким из классов взаимодействовать, и знать их детали.

```

public class HelperFacade {
    public static void generateReport(DBTypes dbType, ReportTypes
reportType, String tableName){
        Connection con = null;
        switch (dbType){
            case MYSQL:
                con = MySqlHelper.getMySqlDBConnection();
                MySqlHelper mySqlHelper = new MySqlHelper();
                switch(reportType){
                    case HTML:
                        mySqlHelper.generateMySqlHTMLReport(tableName, con);
                    break;
                    case PDF:
                        mySqlHelper.generateMySqlPDFReport(tableName, con);
                    break;
                }
                break;
            case ORACLE:
                con = OracleHelper.getOracleDBConnection();
                OracleHelper oracleHelper = new OracleHelper();
                switch(reportType){
                    case HTML:
                        oracleHelper.generateOracleHTMLReport(tableName, con);
                    break;
                    case PDF:
                        oracleHelper.generateOraclePDFReport(tableName, con);
                    break;
                }
                break;
        }
    }
    public static enum DBTypes{
        MYSQL,ORACLE;
    }
    public static enum ReportTypes{
        HTML,PDF;
    }
}

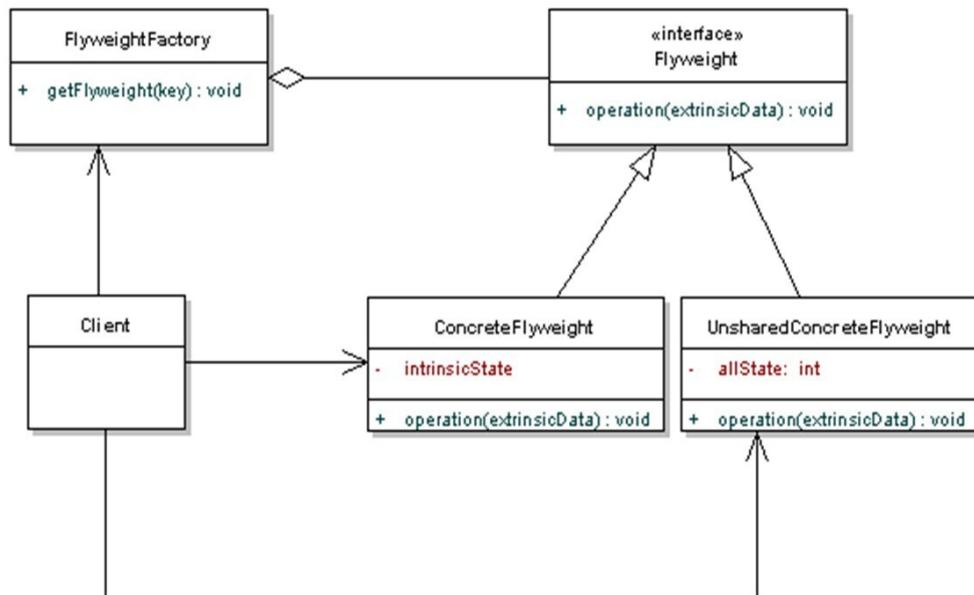
```

Поэтому мы создаем дополнительный класс, с которым и нужно будет взаимодействовать клиенту, просив указав тип базы данных, с которой ему нужно работать.

Всю остальную работу по взаимодействию с конкретными классами возьмет на себя фасад.

Flyweight Pattern

Следующий шаблон, который мы рассмотрим, это Flyweight или Приспособленец.



Паттерн Flyweight в основном используется для уменьшения количества создаваемых объектов и уменьшения объема памяти и повышения производительности.

Шаблон Flyweight пытается повторно использовать уже существующие аналогичные объекты, сохраняя их, и создавая новый объект, если не найден соответствующий объект.

Шаблон Flyweight используется для эффективного совместного использования большого количества объектов.

Примером использования шаблона Flyweight является представление символа в текстовом редакторе.

Вместо того, чтобы каждый символ содержал информацию о шрифте и форматировании, каждый символ может иметь ссылку на объект flyweight, общий для каждого экземпляра одного и того же символа в документе.

В этом случае символу нужно сохранять только его позицию в документе, а не всю информацию о шрифте и форматировании.

При использовании этого шаблона нужно разделить внутренние и внешние данные объекта.

Внутренние данные – это данные, которые делают этот объект уникальным.

А внешние данные – это информация, которая может передаваться через аргументы.

Таким образом, если вы можете сделать некоторые данные объекта внешними для случаев, когда у вас есть большое количество объектов, можно применить шаблон Flyweight.

Интерфейс Flyweight объявляет методы, которые могут принимать и использовать внешние данные.

Класс FlyweightFactory несет ответственность за создание и управление flyweight объектами.

Если нужный объект flyweight еще не создан, FlyweightFactory создаст и вернет его.

В противном случае он вернет объект из текущего пула объектов flyweight.

Класс ConcreteFlyweight добавляет возможности для внутреннего состояния.

Этот объект должен быть совместно используемым.

Класс UnsharedConcreteFlyweight обеспечивает применение этого шаблона без совместного использования.

Тем не менее, большинство применений этого шаблона включают в себя совместно используемые объекты flyweight.

В качестве примера нарисуем линии разных цветов.

Мы будем избегать создания новой линии для каждого цвета, а вместо этого будем повторно использовать линии с одним цветом.

```
//Flyweight
public interface LineFlyweight{
    public Color getColor();
    public void draw(Point location);
}

//ConcreteFlyweight
public class Line implements LineFlyweight{
    private Color color;
    public Line(Color c){
        color = c;
    }
    public Color getColor(){
        return color;
    }
    public void draw(Point location){
        //draw the character on screen
    }
}

//Flyweight factory
public class LineFlyweightFactory{
    private List<LineFlyweight> pool;
    public LineFlyweightFactory(){
        pool = new ArrayList<LineFlyweight>();
    }
    public LineFlyweight getLine(Color c){
        //check if we've already created a line with this color
        for(LineFlyweight line: pool){
            if(line.getColor().equals(c)){return line;
        }
        //if not, create one and save it to the pool
        LineFlyweight line = new Line(c);
        pool.add(line);
        return line;
    }
}
```

Во-первых, мы создадим интерфейс для наших объектов flyweight.

Метод draw получает внешние данные о том, где рисовать линию.

Далее линия реализовывает этот интерфейс.

А фабрика будет управлять созданием объектов линии.

Здесь создается пул линий.

И метод getLine, принимая в качестве аргумента цвет, будет создавать новый объект только в том случае, если в пуле объектов не найдется объекта с данным цветом.

```
LineFlyweightFactory factory = new LineFlyweightFactory();
.....
LineFlyweight line = factory.getLine(Color.RED);
LineFlyweight line2 = factory.getLine(Color.RED);
//can use the lines independently
line.draw(new Point(100, 100));
line2.draw(new Point(200, 100));
```

Клиентский код создает экземпляр фабрики и с помощью фабрики создает объекты линий, которые затем рисует в указанной точке.

Proxy Pattern

Следующий шаблон, который мы рассмотрим, это Ргоху или Заместитель.

Бывают случаи, когда проще, безопаснее или удобнее использовать представителя для представления чего-то.

Представители могут быть отправлены для выступления от имени компании на конференции.

Вместо реальных людей для краш-тестов транспортных средств используются манекены.

А кредитная карта является приемлемой формой оплаты вместо наличных денежных средств.

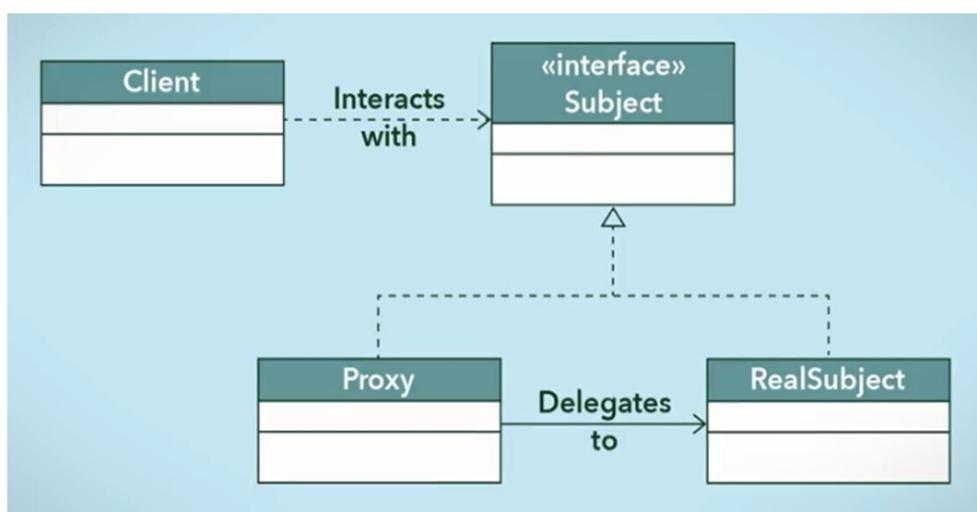
В каждой из этих ситуаций объект представлен прокси-объектом.

Прокси действует как упрощенная или облегченная версия исходного объекта.

Прокси-объект по-прежнему способен выполнять те же задачи, но может делегировать запросы к исходному объекту для их выполнения.

И мы сталкиваемся с аналогичными задачами в программных системах, где вместо оригинала лучше использовать прокси-объект.

Это реализуется с помощью шаблона проектирования прокси, который позволяет прокси-классу представлять класс.



В этом шаблоне проектирования прокси-класс обертывает реальный класс, то есть скрывает ссылку на экземпляр реального класса.

Настоящий класс является частью системы, который, например, может содержать конфиденциальную информацию или будет ресурсоемким для создания экземпляра.

Так как класс proxy действует как оболочка, классы клиентов будут взаимодействовать именно с ним вместо реального класса.

Почему мы хотим использовать прокси-класс?

Три наиболее распространенных сценария – это чтобы действовать как виртуальный прокси, когда прокси-класс используется вместо реального класса, который является ресурсоемким для создания экземпляра.

Этот сценарий работает, например, в случае использования изображений, так как изображение высокой четкости может быть большим.

Постоянная загрузка таких изображений может израсходовать ресурсы вашей системы.

Второй сценарий, это когда прокси обеспечивает защиту или контроль доступа к реальному классу.

Например, защищать контент и позволять к нему доступ только для определенных пользователей.

И третий сценарий, чтобы выступать в качестве удаленного прокси, где прокси-класс является локальным, а реальный класс существует удаленно.

Например, при работе с документом Google, ваш веб-браузер имеет все необходимые ему локальные объекты, которые также существуют на сервере Google.

Прокси-класс обертывает и может делегировать или перенаправлять вызовы реальный класс.

Однако не все вызовы получают делегирование, потому что прокси-класс должен действовать как более легкая версия реального класса и может выполнять некоторые из своих обязанностей.

Прокси-класс будет перенаправлять только самые существенные запросы в реальный класс.

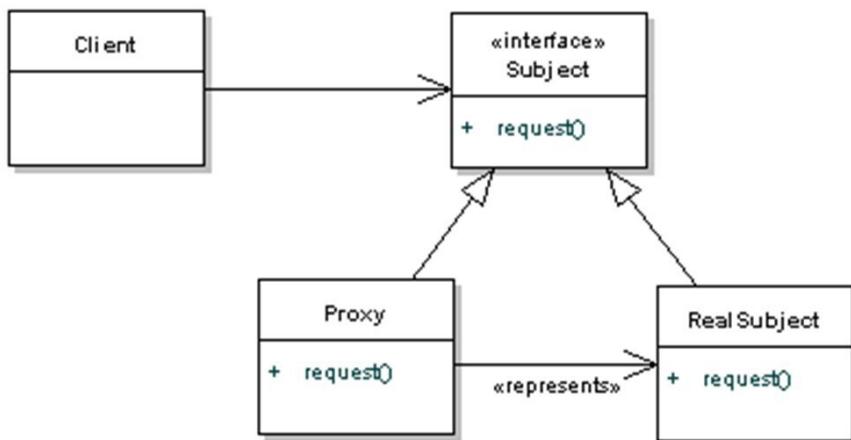
Какие вызовы необходимо поддерживать в прокси-классе?

Поскольку класс proxy предназначен для реального класса, он должен предлагать те же методы.

Это реализуется с помощью общего интерфейса, что также допускает полиморфизм.

Поскольку прокси-сервер и реальный класс являются подтипами общего типа, клиентский класс может взаимодействовать с прокси, который будет иметь тот же ожидаемый интерфейс, что и реальный объект.

И прокси-класс, и реальный класс будут реализовывать интерфейс по-разному.



Таким образом, в шаблоне Прокси, класс представляет функциональность другого класса.

Шаблон Прокси обеспечивает контроль доступа к оригинальному объекту, обеспечивая защиту оригинального объекта от внешнего мира.

Этот шаблон может использоваться в сценарии виртуального прокси-сервера.

Рассмотрим ситуацию, когда есть множество запросов к базе данных для извлечения изображения большого размера.

Так как это дорогостоящая операция, здесь мы можем использовать шаблон прокси-сервера, который будет создавать несколько прокси-серверов и указывать на объект, потребляющий огромные объемы памяти, для дальнейшей обработки.

Реальный объект будет создаваться только при первом запросе клиента.

После этого мы можем просто перенаправлять запрос прокси для повторного использования объекта.

Это позволит избежать дублирования объекта и, следовательно, сохранит память.

Этот шаблон также может использоваться в сценарии защитного прокси.

Он будет действовать на уровне авторизации, чтобы гарантировать, что только авторизованный пользователь имеет доступ к соответствующему контенту.

Например, прокси-сервер, который обеспечивает ограничение доступа в Интернет только для авторизованных пользователей.

Этот шаблон также можно использовать в сценарии удаленного прокси, который работает как заглушка при удаленном вызове.

Удаленный прокси обеспечивает локальное представление объекта, который присутствует на другом адресе.

Этот шаблон также может использоваться в сценарии интеллектуального прокси, который обеспечивает дополнительный уровень безопасности, добавляя определенные действия при обращении к объекту.

Например, чтобы проверить, заблокирован ли реальный объект или нет, прежде чем обращаться к нему.

Рассмотрим использование этого шаблона на примере загрузки изображения большого размера.

```

public interface Image {
    void display();
}

public class ReallImage implements Image {
    private String fileName;

    public ReallImage(String fileName){
        this.fileName = fileName;
        loadFromDisk(fileName);
    }

    @Override
    public void display(){
        System.out.println("Displaying " + fileName);
    }

    private void loadFromDisk(String fileName){
        System.out.println("Loading " + fileName);
    }
}

public class ProxyImage implements Image{
    private ReallImage reallImage;
    private String fileName;

    public ProxyImage(String fileName){
        this.fileName = fileName;
    }

    @Override
    public void display(){
        if(reallImage == null){
            reallImage = new ReallImage(fileName);
        }
        reallImage.display();
    }
}

```

Здесь у нас есть интерфейс показа изображения и реальный объект, который при создании загружает изображение и может его показать.

Чтобы каждый раз не загружать изображение, мы можем создать прокси, который также реализует интерфейс изображения и содержит ссылку на реальный объект.

При вызове метода интерфейса, прокси сначала проверяет, создан ли реальный объект.
Если реальный объект уже существует, прокси вызывает его метод интерфейса.
Если объекта нет, прокси создает новый объект, и затем вызывает его метод интерфейса.

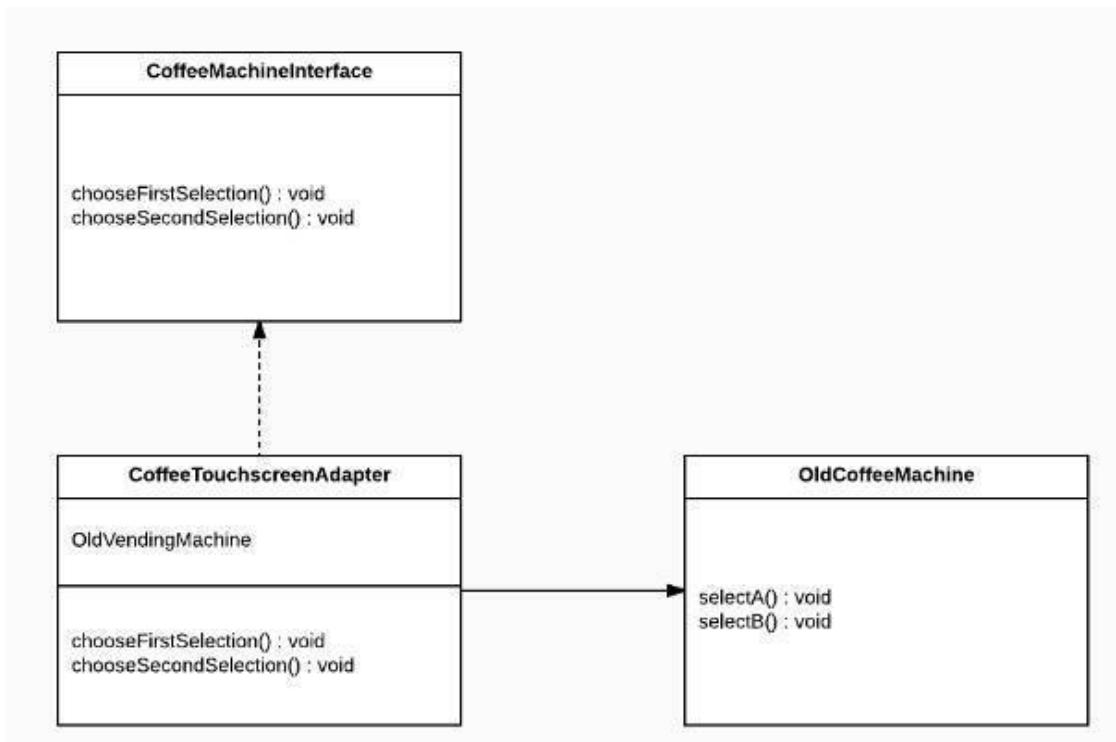
Задание

Вы работаете в офисе со старой кофемашиной, которая делает два разных кофе.

Новый босс хочет купить новую кофемашину с сенсорным экраном, который также может подключаться к старой кофемашине.

Дополните прилагаемый код, чтобы добавить адаптер, так, чтобы новый сенсорный экран работал со старой кофемашиной.

Используйте следующую диаграмму классов UML:



CoffeeMachineInterface.java

```
public interface CoffeeMachineInterface {  
}
```

OldCoffeeMachine.java

```
public class OldCoffeeMachine {  
}
```

CoffeeTouchscreenAdapter.java

```
public class CoffeeTouchscreenAdapter implements CoffeeMachineInterface {  
}
```

Ответ

CoffeeMachineInterface.java

```
public interface CoffeeMachineInterface {  
    public void chooseFirstSelection();  
    public void chooseSecondSelection();  
}
```

OldCoffeeMachine.java

```
public class OldCoffeeMachine {
```

```
public void selectA() {  
    System.out.println("A – Selected");  
}  
Public void selectB() {  
    System.out.println("B – Selected");  
}  
}  
CoffeeTouchscreenAdapter.java  
public class CoffeeTouchscreenAdapter implements CoffeeMachineInterface {  
  
    OldCofffeeMachine theMachine;  
  
    public OldCoffeeMachineAdapter(OldCoffeeMachine newMachine) {  
        theMachine = newMachine;  
    }  
  
    public void chooseFirstSelection() {  
        theMachine.selectA();  
    }  
  
    public void chooseSecondSelection() {  
        theMachine.selectB();  
    }  
}
```

Задание

Вас попросили создать приложение для воспроизведения плейлистов.

Мы будем предполагать, что каждый плейлист может состоять из песен или других плейлистов или их комбинации.

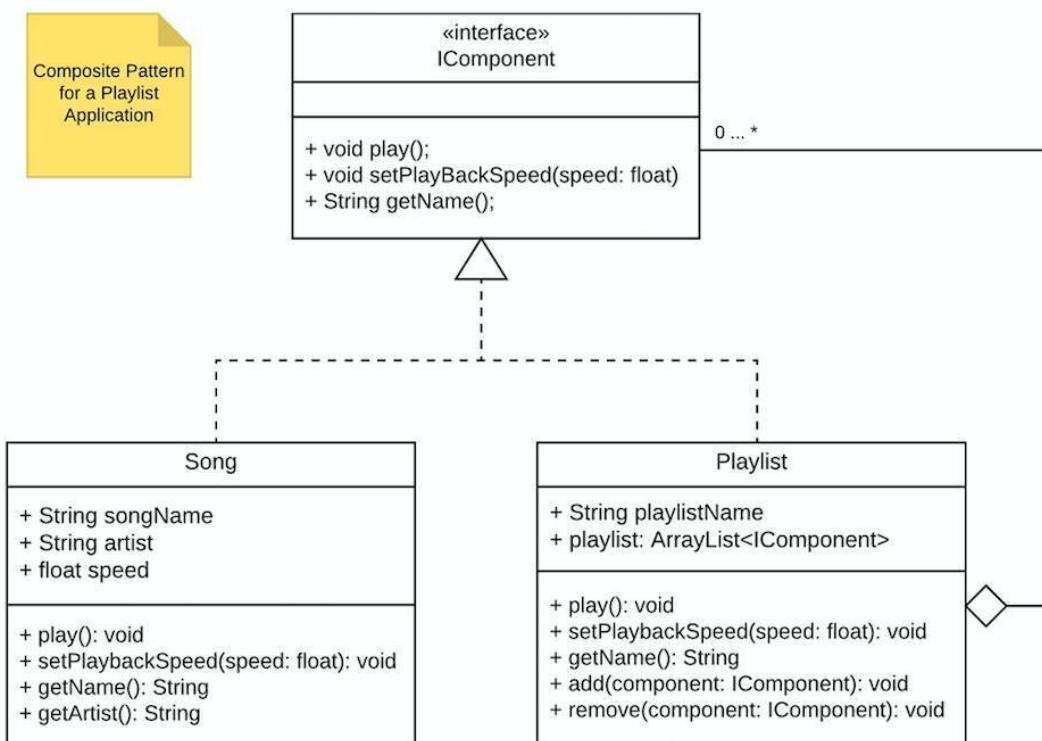
Ваш руководитель проекта сказал вам, что лучше всего использовать в этой ситуации шаблон Composite.

Следующая диаграмма классов UML связывает объекты и отношения приложения с использованием шаблона Composite.

В этом задании вам необходимо дополнить предоставленный код.

Примечание. За исключением класса Playlist, вам не нужно реализовывать методы, просто напишите комментарии (например, // play song).

В классе Playlist напишите метод добавления песен в список воспроизведения.



[Program.java]

```
public class Program {
    public static void main(String args[]) {
        // Make new empty "Study" playlist
        Playlist studyPlaylist = new Playlist("Study");
        // Make "Synth Pop" playlist and add 2 songs to it.
        Playlist synthPopPlaylist = new Playlist("Synth Pop");
        Song synthPopSong1 = new Song("Girl Like You", "Toro Y Moi" );
        Song synthPopSong2 = new Song("Outside", "TOPS");
```

```

synthPopPlaylist.add(synthPopSong1);
synthPopPlaylist.add(synthPopSong2);
// Make "Experimental" playlist and add 3 songs to it,
// then set playback speed of the playlist to 0.5x
Playlist experimentalPlaylist = new Playlist("Experimental");
Song experimentalSong1 = new Song("About you", "XXYYXX");
Song experimentalSong2 = new Song("Motivation", "Clams Casino");
Song experimentalSong3 = new Song("Computer Vision", "Oneohtrix Point Never");
experimentalPlaylist.add(experimentalSong1);
experimentalPlaylist.add(experimentalSong2);
experimentalPlaylist.add(experimentalSong3);
float slowSpeed = 0.5f;
experimentalPlaylist.setPlaybackSpeed(slowSpeed);
// Add the "Synth Pop" playlist to the "Experimental" playlist
experimentalPlaylist.add(synthPopPlaylist);
// Add the "Experimental" playlist to the "Study" playlist
studyPlaylist.add(experimentalPlaylist);
// Create a new song and set its playback speed to 1.25x, play this song,
// get the name of glitchSong → "Textuell", then get the artist of this song → "Oval"
Song glitchSong = new Song("Textuell", "Oval");
float fasterSpeed = 1.25f;
glitchSong.setPlaybackSpeed(fasterSpeed);
glitchSong.play();
String name = glitchSong.getName();
String artist = glitchSong.getArtist();
System.out.println ("The song name is " + name );
System.out.println ("The song artist is " + artist );
// Add glitchSong to the "Study" playlist
studyPlaylist.add(glitchSong);
// Play "Study" playlist.
studyPlaylist.play();
// Get the playlist name of studyPlaylist → "Study"
System.out.println ("The Playlist's name is " + studyPlaylist.getName() );
}
}

```

[IComponent.java]

```

public interface IComponent {
// Your code goes here!
}

```

[Playlist.java]

```

public class Playlist implements IComponent {
public String playlistName;
public ArrayList<IComponent> playlist = new ArrayList();
public Playlist(String playlistName) {
this.playlistName = playlistName;
}
// Your code goes here!
}

```

[Song.java]

```
public class Song implements IComponent {  
    public String songName;  
    public String artist;  
    public float speed = 1; // Default playback speed  
    public Song(String songName, String artist) {  
        this.songName = songName;  
        this.artist = artist;  
    }  
    // Your code goes here!  
}
```

Ответ

[Program.java]

```
public class Program {  
    public static void main(String args[]) {  
        // Make new empty "Study" playlist  
        Playlist studyPlaylist = new Playlist("Study");  
        // Make "Synth Pop" playlist and add 2 songs to it.  
        Playlist synthPopPlaylist = new Playlist("Synth Pop");  
        Song synthPopSong1 = new Song("Girl Like You", "Toro Y Moi");  
        Song synthPopSong2 = new Song("Outside", "TOPS");  
        synthPopPlaylist.add(synthPopSong1);  
        synthPopPlaylist.add(synthPopSong2);  
        // Make "Experimental" playlist and add 3 songs to it,  
        // then set playback speed of the playlist to 0.5x  
        Playlist experimentalPlaylist = new Playlist("Experimental");  
        Song experimentalSong1 = new Song("About you", "XXYYXX");  
        Song experimentalSong2 = new Song("Motivation", "Clams Casino");  
        Song experimentalSong3 = new Song("Computer Vision", "Oneohtrix Point Never");  
        experimentalPlaylist.add(experimentalSong1);  
        experimentalPlaylist.add(experimentalSong2);  
        experimentalPlaylist.add(experimentalSong3);  
        float slowSpeed = 0.5f;  
        experimentalPlaylist.setPlaybackSpeed(slowSpeed);  
        // Add the "Synth Pop" playlist to the "Experimental" playlist  
        experimentalPlaylist.add(synthPopPlaylist);  
        // Add the "Experimental" playlist to the "Study" playlist  
        studyPlaylist.add(experimentalPlaylist);  
        // Create a new song and set its playback speed to 1.25x, play this song,  
        // get the name of glitchSong → "Textuell", then get the artist of this song → "Oval"  
        Song glitchSong = new Song("Textuell", "Oval");  
        float fasterSpeed = 1.25f;  
        glitchSong.setPlaybackSpeed(fasterSpeed);  
        glitchSong.play();  
        String name = glitchSong.getName();  
        String artist = glitchSong.getArtist();  
        System.out.println("The song name is " + name);  
        System.out.println("The song artist is " + artist);  
        // Add glitchSong to the "Study" playlist  
        studyPlaylist.add(glitchSong);
```

```
// Play "Study" playlist.  
studyPlaylist.play();  
  
// Get the playlist name of studyPlaylist → "Study"  
System.out.println ("The Playlist's name is " + studyPlaylist.getName() );  
}  
}  
[IComponent.java]  
public interface IComponent {  
void play();  
void setPlaybackSpeed(float speed);  
String getName();  
}  
[Playlist.java]  
public class Playlist implements IComponent {  
public String playlistName;  
public ArrayList<IComponent> playlist = new ArrayList();  
public Playlist(String playlistName) {  
this.playlistName = playlistName;  
}  
public void add(IComponent component) {  
playlist.add(component);  
}  
public void remove(IComponent component) {  
playlist.remove(component);  
}  
public void play(){  
for(IComponent component : playlist) {  
component.play();  
}  
}  
public void setPlaybackSpeed(float speed) {  
for(IComponent component : this.playlist){  
component.setPlaybackSpeed(speed);  
}  
}  
public String getName() {  
return this.playlistName;  
}  
}  
[Song.java]  
public class Song implements IComponent {  
public String songName;  
public String artist;  
public float speed = 1; // Default playback speed  
public Song(String songName, String artist ) {  
this.songName = songName;  
this.artist = artist;  
}
```

```
public void play() {  
    // Play the song using this.speed  
}  
public void setPlaybackSpeed(float speed) {  
    this.speed = speed;  
}  
public String getName() {  
    return this.songName;  
}  
public String getArtist() {  
    return this.artist;  
}  
}
```

Вопросы

Вопрос 1

Когда лучше всего использовать шаблоны проектирования?

Для задачи, которая уникальна для вашей программы.

Для часто встречающейся задачи. +

Для устранения кода спагетти.

Объясняя решение своим коллегам-разработчикам. +

Вопрос 2

Какова цель шаблона Singleton?

для принудительного создания экземпляра только одного объекта класса +

для обеспечения простых классов только одним методом

обеспечить глобальный доступ к объекту +

для обеспечения взаимодействия класса только с одним другим классом

Вопрос 3

Что значит «разрешить подкласс» в шаблоне Factory Method?

Подкласс определяет методы для конкретного экземпляра. Таким образом, тип объекта определяется тем, какой создается подкласс. +

Подкласс решает, какой объект создавать, но вызывает метод, который определен в суперклассе для создания экземпляра класса.

Подкласс передает параметр в фабрику, которая определяет, какой объект создается.

Вопрос 4

Что мы называем созданием объекта, например, с оператором new в Java?

Конкретный экземпляр +

Реализация объекта

Создание класса

Объявление

Вопрос 5

Каковы преимущества шаблона фасада?

Класс Facade перенаправляет запросы по мере необходимости. +

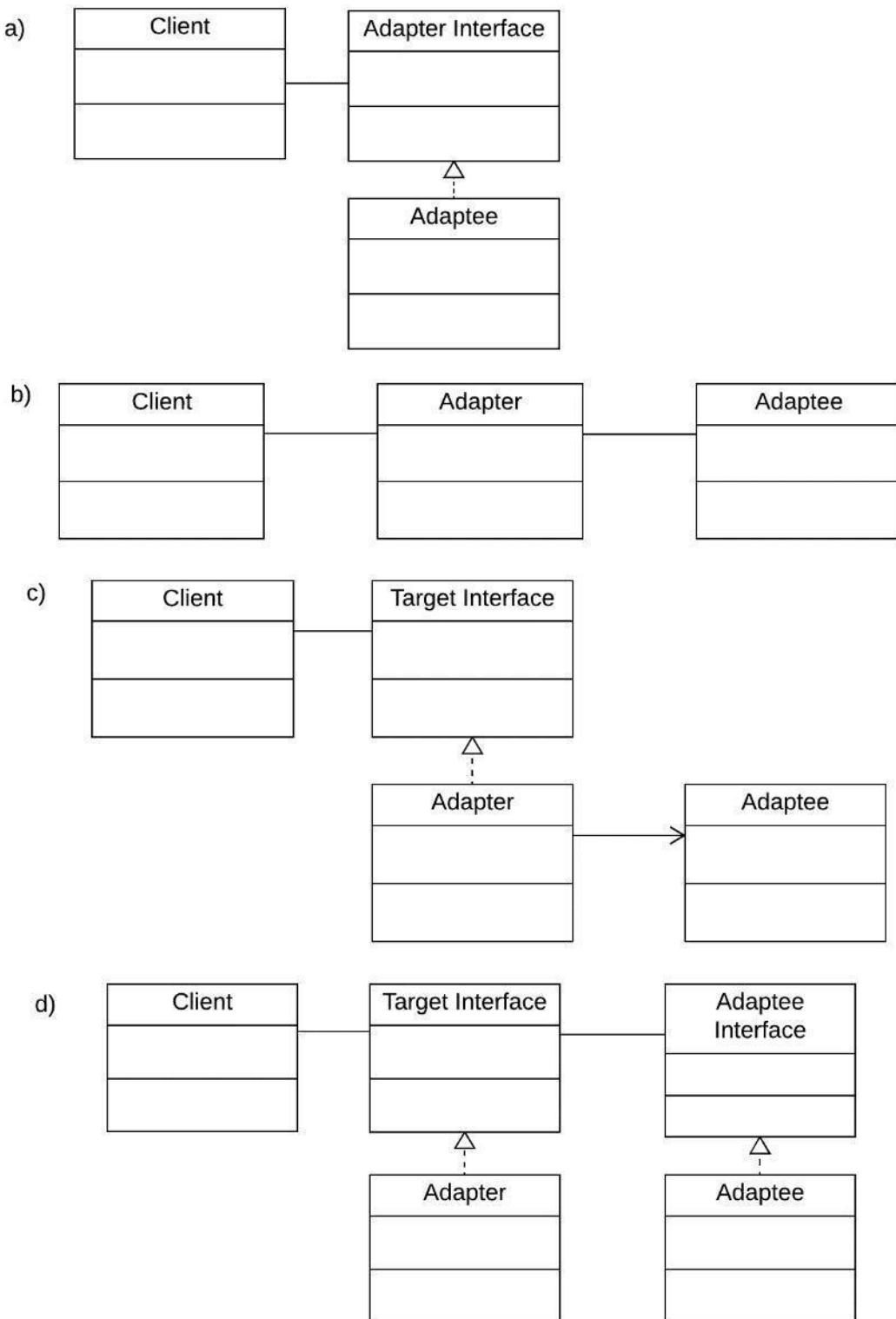
Клиент и подсистема более слабо связаны. +

Подсистема может обрабатывать больше клиентов.

Сложность подсистемы скрыта. +

Вопрос 6

На какой из следующих диаграмм показан шаблон адаптера?



Ответ с).

Вопрос 7

Как работает шаблон декоратора?

добавление функций в класс с новым классом

**создает поведение путем расположения объектов в стеке +
расширяет методы класса с наследованием**

инкапсулирует класс, чтобы дать ему другой интерфейс

Вопрос 8

Каковы типы объектов, которые используются в композитном шаблоне?

branch

leaf +

trunk

root

composite +

Вопрос 9

Как вы принудительно создадите только один объект Singleton?

Напишите метод, который может создать новый объект Singleton или вернуть существующий. +

Укажите в комментариях, что должен быть создан только один объект Singleton.

Выбросить исключение, если объект Singleton уже создан.

Дайте классу Singleton приватный конструктор. +

Поведенческие шаблоны проектирования. Chain Of Responsibility Pattern

Следующая группа шаблонов – это поведенческие шаблоны проектирования.

Поведенческие шаблоны проектирования связаны с взаимодействием и ответственностью объектов.

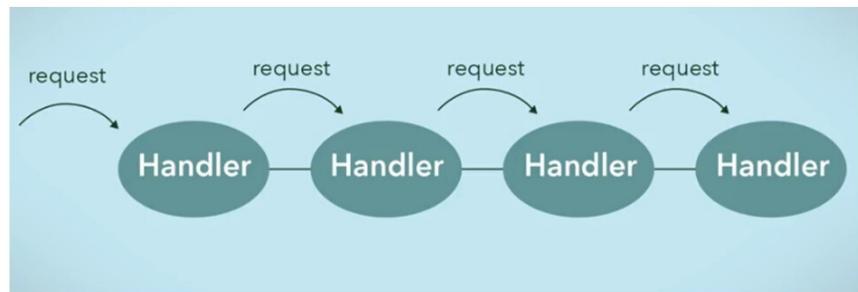
В этих шаблонах проектирования взаимодействие между объектами должно быть таким, чтобы они могли легко взаимодействовать друг с другом, и при этом все еще должны быть слабо связаны между собой.

Это означает, что реализация и клиент должны быть слабо связаны, чтобы избежать жесткого кодирования и зависимостей.

Первый шаблон в этой группе, который мы рассмотрим – это Цепочка обязанностей.

Цепочка обязанностей – это именно то, как это звучит.

Это цепочка объектов, которые отвечают за обработку запросов.



Например, вы обращаетесь за помощью с проблемой со здоровьем.

Сначала вы посещаете своего врача, но ваш случай необычен, поэтому вас направляют к специалисту.

Но специалист находится в отпуске и не может принять вас принять, и поэтому вы обращаетесь к другому специалисту.

И, наконец, этот специалист справляется с вашей проблемой.

Вам все равно, кто на самом деле поможет в этой цепочке медицинских работников.

В разработке программного обеспечения цепочка ответственности представляет собой серию объектов-обработчиков, которые связаны друг с другом.

У этих обработчиков есть методы, которые предназначены для обработки определенных запросов.

Когда клиентский объект отправляет запрос, первый обработчик в цепочке будет пытаться его обработать.

Если обработчик сможет обработать запрос, запрос заканчивается на этом обработчике.

Если обработчик не сможет обработать запрос, обработчик отправит запрос следующему обработчику в цепочке, который попытается обработать запрос.

Опять же, если обработчик не сможет обработать запрос, он отправит запрос следующему обработчику.

И эта передача запроса будет продолжаться до тех пор, пока мы не найдем обработчик, который сможет обработать запрос.

Если запрос пройдет через всю цепочку обработчиков, и обработчики не смогут обработать его, запрос не будет выполнен.

Вы также можете подумать об этом шаблоне, по аналогии с обработкой исключений в Java.

При обработке исключений вы пишете ряд блоков try-catch, чтобы гарантировать, что исключения обрабатываются должным образом.

Когда возникает исключение, ожидается, что один из блоков catch обработает его.

Где вы можете использовать этот шаблон дизайна в своем программном обеспечении?

Цепочка может использоваться для различных целей.

Предположим, вы настраиваете почтовую службу, где есть множество способов фильтрации электронной почты.

Вы можете создавать объекты, которые будут работать как отдельные фильтры.

У каждого из этих объектов фильтра будет метод.

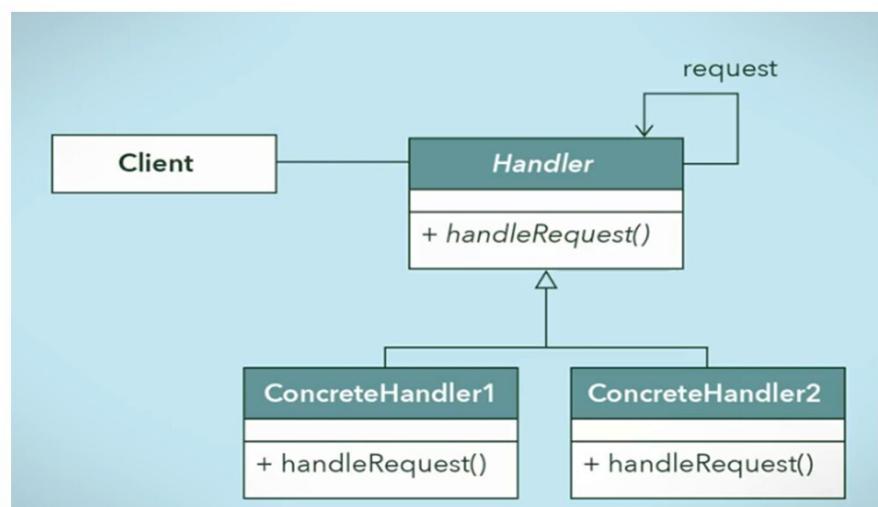
И этот метод будет проверять, соответствует ли содержимое сообщений электронной почты конкретному правилу фильтра.

Если содержимое будет соответствовать правилу, оно будет обрабатываться фильтром.

Например, объект фильтра может поместить это сообщение в папку спама.

Если содержимое не будет соответствовать правилу, фильтр вызовет следующий фильтр в цепочке для обработки запроса.

Использование шаблона проектирования Chain of Responsibility является очень распространенным способом настройки такого поведения.



В общем, диаграмма классов UML для шаблона проектирования Chain of Responsibility выглядит следующим образом.

Все объекты в цепочке являются обработчиками, которые реализуют общий метод обработки запроса, объявленный в абстрактном суперклассе обработчика.

Эти объекты обработчика соединены от одного к другому в цепочке.

И существуют подклассы обработчика для обработки определенных запросов.

Вы видите проблему с дизайном?

Что, если, например, во втором фильтре есть ошибка?

Что если не будет передан запрос на следующий фильтр?

И обработка закончится преждевременно.

Это проблема с цепочкой обязанностей.

Нам нужен алгоритм, чтобы каждый класс фильтра обрабатывал запросы аналогичным образом.

То есть мы хотим убедиться, что каждый фильтр проходит следующие шаги.

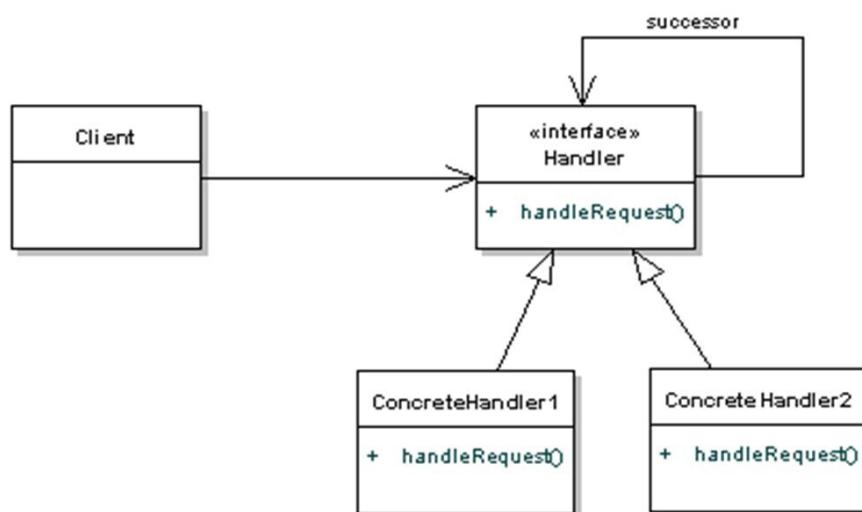
Сначала проверить соответствие правилу.

Если соответствие есть, сделать что-то конкретное.

Если соответствия нет, нужно вызвать следующий фильтр в списке.

Для этого вы можете использовать шаблон Template, который мы рассмотрим позже.

Часто шаблоны проектирования могут иметь свои собственные внутренние проблемы, и вы можете использовать другие шаблоны проектирования для устранения этих проблем.



Таким образом, как следует из названия, шаблон Цепочки создает цепочку объектов-приемников для запроса.

Этот шаблон отделяет отправителя и получателя от запроса в зависимости от типа запроса.

В этом шаблоне обычно каждый приемник содержит ссылку на другой приемник.

Если один объект не может обработать запрос, запрос переходит к следующему приемнику и так далее.

На слайде Handler определяет интерфейс, необходимый для обработки запроса, в то время как классы ConcreteHandler обрабатывают запросы, за которые они отвечают.

Если какой-то ConcreteHandler не может обработать запрос, он передает запрос его приемнику, на который он содержит ссылку.

Объектам в цепочке просто нужно знать, как перенаправить запрос другим объектам.

И вы можете динамически изменить эту цепочку во время выполнения.

Этот шаблон используется в Windows для обработки событий, генерируемых с клавиатуры или мыши.

Системы обработки исключений также реализуют этот шаблон, когда среда выполнения проверяет, предоставлен ли обработчик для исключения в стеке вызовов.

Если обработчик не определен, исключение приведет к сбою в программе, так как оно будет не обработано.

```
public interface Chain {  
    public abstract void setNext(Chain nextInChain);  
    public abstract void process(Request request);  
}  
  
public class NegativeProcessor implements Chain {  
    private Chain nextInChain;  
  
    public void setNext(Chain c) {  
        nextInChain = c;  
    }  
  
    public void process(Request request) {  
        if (request.getNumber() < 0) {  
            System.out.println("NegativeProcessor: " +  
                request.getNumber());  
        } else {  
            nextInChain.process(request);  
        }  
    }  
}  
  
public class PositiveProcessor implements Chain {  
    private Chain nextInChain;  
  
    public void setNext(Chain c) {  
        nextInChain = c;  
    }  
  
    public void process(Request request) {  
        if (request.getNumber() >= 0) {  
            System.out.println("PositiveProcessor: " +  
                request.getNumber());  
        } else {  
            nextInChain.process(request);  
        }  
    }  
}
```

В этом примере у нас есть интерфейс, который обрабатывает запрос и устанавливает преемника в цепочке.

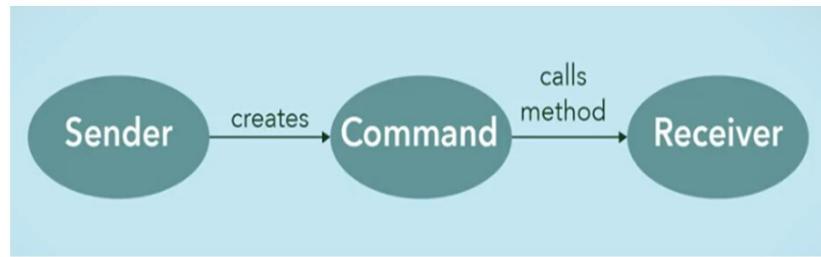
Классы, реализующие интерфейс, при обработке запроса проверяют, удовлетворяет ли он определенному требованию.

И если запрос не подходящий, вызывается метод преемника.

Таким образом запрос передается следующему объекту в цепочке.

Command Pattern

Следующий шаблон, который мы рассмотрим, это Команда или Действие.



Шаблон команды инкапсулирует запрос как собственный объект.

Обычно, когда один объект запрашивает второй объект для выполнения действия, первый объект будет вызывать метод второго объекта, а второй объект выполнит задачу.

И в этой ситуации объект-отправитель напрямую должен связываться с объектом-получателем.

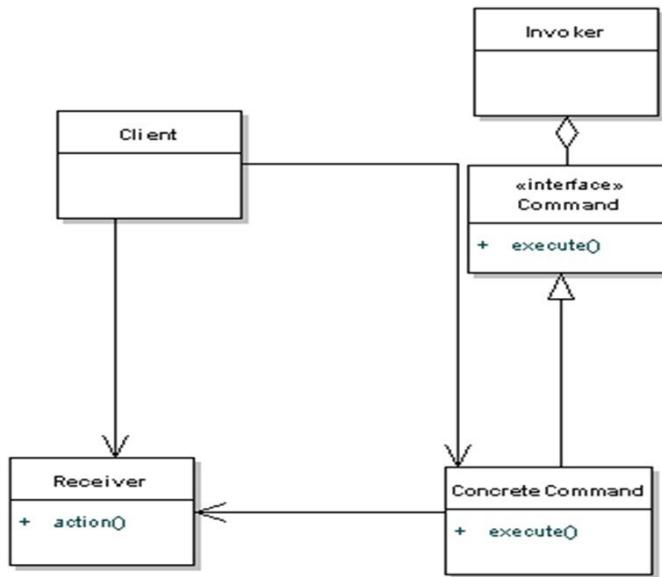
Вместо того, чтобы эти объекты непосредственно связывались друг с другом, шаблон команды создает объект команды между отправителем и получателем.

Таким образом, отправителю не нужно знать о получателе и методах вызова.

Таким образом, объект-отправитель создает объект команды.

Но что на самом деле заставляет объект команды делать то, что он должен делать, и вызывать конкретный объект-получатель для выполнения задачи?

Здесь приходит invoker.



В шаблоне команды есть еще один объект, который вызывает объекты команды для выполнения задачи, называемый invoker.

Здесь можно использовать менеджера команд, который отслеживает команды, манипулирует ими и вызывает их.

Где вы можете реализовать шаблон команды в своем программном обеспечении?

Для использования шаблона команды существует много разных целей.

Одной из целей использования шаблона команды является хранение и планирование различных запросов.

Когда объект вызывает метод другого объекта, вы не можете ничего делать с вызовами метода.

Превращение различных запросов в вашем программном обеспечении в объекты команды может позволить вам рассматривать их как способ обработки других объектов.

Вы можете хранить эти объекты команд в списках, вы можете манипулировать ими до их завершения или вы можете поместить их в очередь, чтобы вы могли планировать выполнение разных команд в разное время.

Например, вы можете использовать шаблон команды для вызова таймера в программном обеспечении.

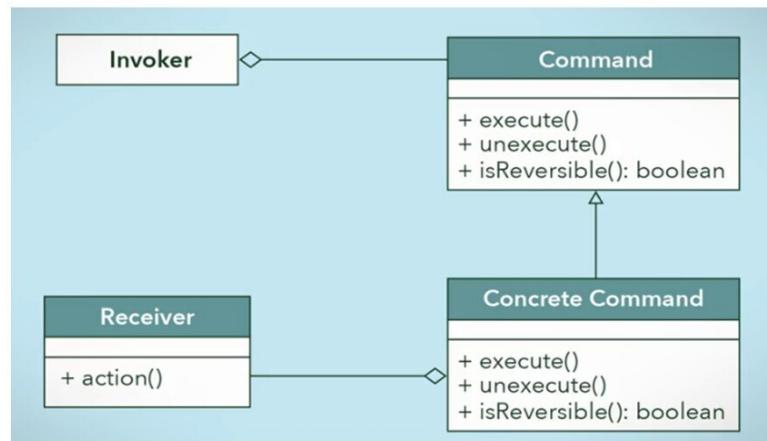
Эта команда может быть помещена в очередь, так что команда может быть завершена позже, когда на самом деле запланировано событие.

Еще одна важная цель шаблона команды – это возможность отменить или перезапустить команды.

Также как в текстовом редакторе, вы можете совершить откат при редактировании текста.

Вы можете использовать шаблон команды, чтобы разрешить повтор / отмену для любого типа приложения.

Как шаблон команды может выглядеть в UML-диаграмме и в исходном коде?



У вас есть суперкласс команды, и все команды будут экземплярами подклассов этого суперкласса команды.

Суперкласс определяет общее поведение ваших команд.

Каждая команда может иметь методы execute, unexecute и isReversible.

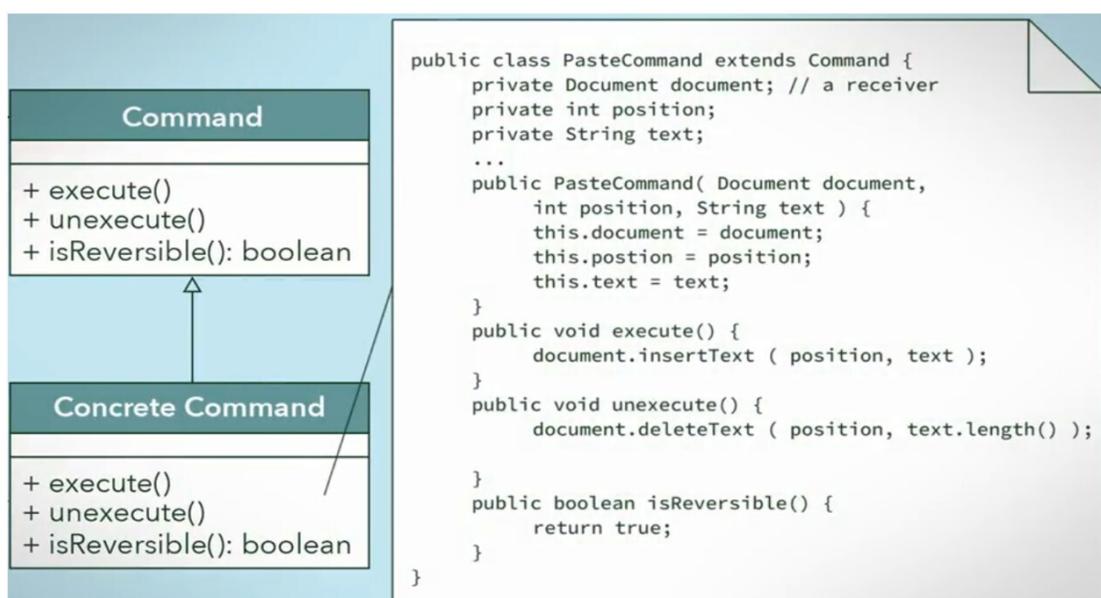
Метод execute выполняет работу, которую должна выполнить команда.

Метод unexecute выполняет работу по отмене команды, а метод isReversible определяет, является ли команда обратимой, и возвращает true, если команда может быть отменена.

Могут быть некоторые команды, которые нельзя отменить.

Например, команда сохранить как, которую не имеет смысла отменять.

Конкретные классы команд вызывают конкретные классы приемников, чтобы выполнить фактическую работу по завершении команды.



Если мы рассмотрим конкретный класс команды, например, вставку текста, мы увидим, каким должен быть объект команды.

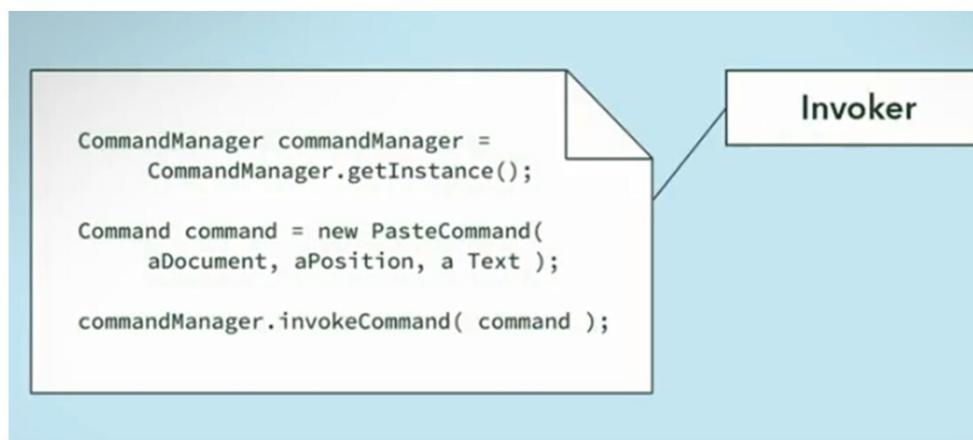
Мы видим, что команда вставки расширяет суперкласс команды.

Объект команды отслеживает, где будет вставлен текст и что будет вставлено.

Это очень важный аспект объектов команды.

Они должны отслеживать множество подробностей о текущем состоянии, чтобы команды были обратимыми.

В этом случае команда вставки также является обратимой командой, потому что вы можете удалить только что вставленный объект.



Исходный код для invoker является довольно простым.

Во-первых, здесь потребуется ссылка на менеджера команд, который является объектом, управляющим историей команд.

Затем invoker создает объект команды с информацией, необходимой для выполнения команды, а затем вызывает менеджера команд для выполнения команды.

Таким образом, шаблон Команды – это шаблон проектирования, в котором запрос обертывается в объект как команда и передается вызывающему объекту.

Вызывающий объект ищет соответствующий объект, который сможет обработать эту команду и передает ему команду для выполнения.

Шаблон команды отделяет объект, который вызывает операцию, от объекта, который фактически выполняет операцию.

Это позволяет легко добавлять новые команды, так как существующие классы остаются неизменными.

Этот шаблон используется, когда нужно параметризовать объекты в соответствии с действием, или, когда нужно создавать и выполнять запросы в разное время, или, когда необходимо поддерживать откаты операций, протоколирование или транзакции.

Шаблон Команды объявляет интерфейс для всех команд, предоставляя простой метод execute, который запрашивает Receiver команды для выполнения операции.

Receiver знает, что делать, чтобы выполнить запрос.

Invoker содержит команду и может заставить команду выполнить запрос, с помощью вызова метода execute.

Клиент создает экземпляр конкретной команды и устанавливает Receiver для команды.

Объект конкретной команды определяет связь между действием и получателем Receiver.

Когда Invoker вызовет метод execute, конкретная команда запустит одно или несколько действий в приемнике Receiver.

```
//Command
public interface Command{
    public void execute();
}

//Concrete Command
public class LightOnCommand implements Command{
    //reference to the light
    Light light;
    public LightOnCommand(Light light){
        this.light = light;
    }
    public void execute(){
        light.switchOn();
    }
}

//Receiver
public class Light{
    private boolean on;
    public void switchOn(){
        on = true;
    }
    public void switchOff(){
        on = false;
    }
}

//Concrete Command
public class LightOffCommand implements Command{
    //reference to the light
    Light light;
    public LightOffCommand(Light light){
        this.light = light;
    }
    public void execute(){
        light.switchOff();
    }
}

//Invoker
public class RemoteControl{
    private Command command;
    public void setCommand(Command command){
        this.command = command;
    }
    public void pressButton(){
        command.execute();
    }
}
```

Здесь у нас есть интерфейс команды, который реализуют два класса, один класс включает свет в приемнике Receiver, а другой выключает свет.

И у нас есть Invoker, который содержит ссылку на объект Command, и его метод pressButton вызывает метод execute объекта Command.

```
RemoteControl control = new RemoteControl();

Light light = new Light();

Command lightsOn = new LightsOnCommand(light);
Command lightsOff = new LightsOffCommand(light);
//switch on
control.setCommand(lightsOn);
control.pressButton();

//switch off
control.setCommand(lightsOff);
control.pressButton();
```

Таким образом, клиент создает объект Invoker. В данном случае это RemoteControl.

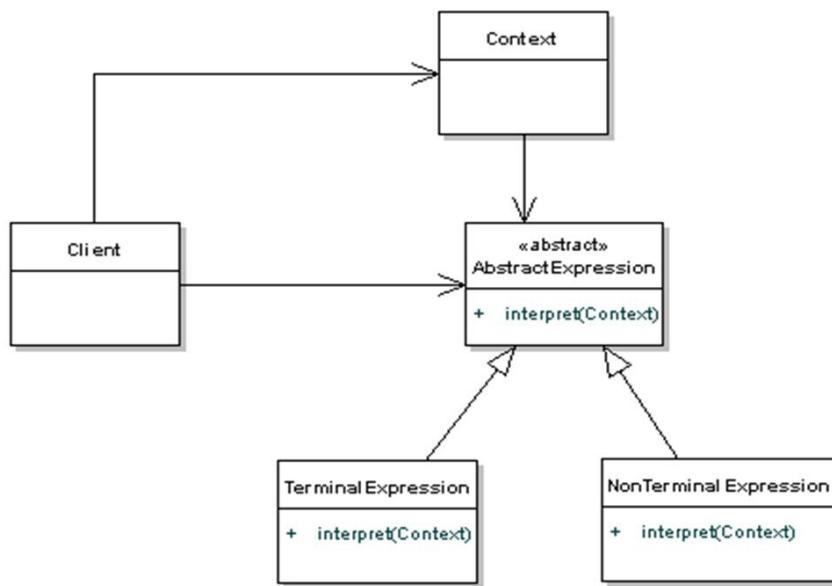
Далее создает объект Receiver. В данном случае это Light.

Создает команды, связывая их с Receiver. И устанавливает команду для Invoker.

После чего вызывает метод Invoker, который запускает действие команды, которая в свою очередь выполняет операцию Receiver.

Interpreter Pattern

Следующий шаблон, который мы рассмотрим, это Интерпретатор.



Шаблон Интерпретатор дает возможность вычислить грамматику или выражение языка.

Этот шаблон предполагает реализацию интерфейса выражения, которая позволяет интерпретировать конкретный контекст.

Этот шаблон используется при разборе SQL выражений, обработке символов и т. д.

Шаблон Интерпретации определяет представление грамматики данного языка и использует это представление для интерпретации выражений языка.

Этот шаблон позволяет легко расширять и изменять не сложную грамматику языка.

На слайде **Context** содержит информацию, которая является глобальной для интерпретатора.

Класс **AbstractExpression** предоставляет интерфейс для выполнения операции.

Класс **TerminalExpression** реализует интерфейс интерпретации, связанный с любыми терминальными выражениями в определенной грамматике.

Клиент определяет абстрактное дерево синтаксиса, которое состоит из выражений **TerminalExpression** и **NonTerminalExpression**.

И Клиент запускает операцию интерпретации.

Обратите внимание, что дерево синтаксиса обычно реализуется с использованием шаблона **Composite**.

Этот шаблон позволяет отделить низлежащие выражения от грамматики.

Шаблон интерпретатора следует использовать, когда у вас есть простая грамматика, которая может быть представлена как абстрактное дерево синтаксиса.

Также этот шаблон может использоваться, когда вам нужна программа для создания различных типов вывода, например, генератора отчетов.

Лучшим примером применения шаблона проектирования Интерпретатор является java-компилятор, который интерпретирует исходный код java в байтовый код, который понятен JVM.

Давайте рассмотрим этот шаблон на примере, в котором пользовательский ввод может конвертироваться в две формы: число в двоичном или число в шестнадцатеричном формате.

```

public class InterpreterContext{
    public String getBinaryFormat(int i){
        return Integer.toBinaryString(i);
    }
    public String getHexadecimalFormat(int i){
        return Integer.toHexString(i);
    }
}

public class IntToBinaryExpression implements Expression {
    private int i;
    public IntToBinaryExpression(int c){
        this.i=c;
    }
    @Override
    public String interpret(InterpreterContext ic) {
        return ic.getBinaryFormat(this.i);
    }
}

public interface Expression {
    String interpret(InterpreterContext ic);
}

public class IntToHexExpression implements Expression {
    private int i;
    public IntToHexExpression(int c){
        this.i=c;
    }
    @Override
    public String interpret(InterpreterContext ic) {
        return ic.getHexadecimalFormat(i);
    }
}

```

В ответ на ввод целого числа, интерпретатор будет возвращать строку, представляющую число в двоичном или шестнадцатеричном формате.

Нашим первым шагом будет написать класс контекста интерпретатора, который будет выполнять фактическую интерпретацию.

Далее нам нужно создать разные типы выражений, которые будут потребляться классом контекста интерпретатора.

Обратите внимание, что класс контекста передается в метод выражения.

Далее мы создаем две реализации выражений, одну для преобразования целого числа в двоичный формат и другую для преобразования целого числа в шестнадцатеричный формат.

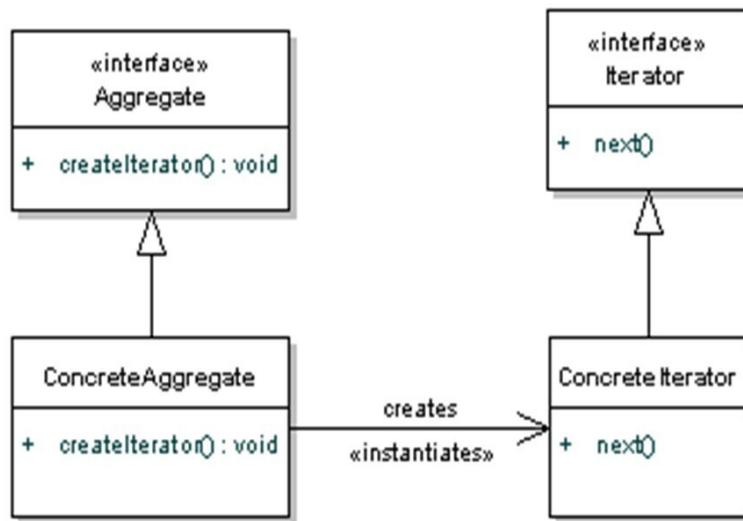
Теперь клиент должен создать объект одного из выражений, передав в конструктор целое число.

Создать объект контекста и вызвать метод `interpret`, передав в него контекст.

В ответ он получит нужную интерпретацию.

Iterator Pattern

Следующий шаблон, который мы рассмотрим, это Итератор или Курсор.



Шаблон Итератор очень часто используется для получения доступа к элементам объекта коллекции последовательно, без необходимости знать ее низлежащее представление.

Интерфейс Aggregate определяет интерфейс для создания объекта Iterator.

Класс ConcreteAggregate реализует этот интерфейс и возвращает экземпляр итератора ConcreteIterator.

Интерфейс Iterator определяет интерфейс для доступа к элементам объекта Aggregate и обхода элементов, а класс ConcreteIterator реализует этот интерфейс, отслеживая текущую позицию в обходе объекта Aggregate.

Java обеспечивает реализацию шаблона Iterator, который предоставляет методы next и hasNext.

Создание итератора в Java обычно выполняется с помощью метода с именем iterator в классе контейнера.

```

List<String> list = new ArrayList<String>();

Iterator it = list.iterator();

while(it.hasNext()){

String s = it.next();

}

```

Вот как работает итератор в списке.

Мы получаем итератор из контейнера и, используя его методы next и hasNext, перемещаемся по списку, ничего не зная о его реализации.

```

public class CollectionofNames implements Container{
    public String name[]={“name1”, “name2”, “name3”};
    @Override
        public Iterator getIterator() {
            return new CollectionofNamesIterate();
        }
    private class CollectionofNamesIterate implements Iterator{
        int i;
        @Override
        public boolean hasNext() {
            if (i<name.length){
                return true;
            }
            return false;
        }
        @Override
        public Object next() {
            if(this.hasNext()){
                return name[i++];
            }
            return null;
        } } }
public interface Container{
    public Iterator getIterator();
}
```

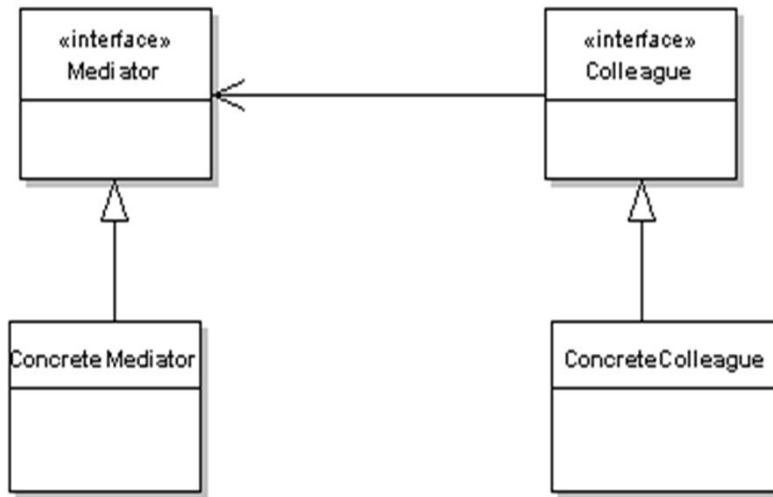
В этом упрощенном примере, у нас есть интерфейс контейнера, который возвращает итератор, и интерфейс итератора, который предоставляет методы next и hasNext.

Далее мы создаем класс, который реализует интерфейс контейнера.

Внутри класса мы создаем реализацию итератора, определяя методы next и hasNext, с помощью которых мы можем перемещаться по вшитому массиву строк.

Mediator Pattern

Следующий шаблон, который мы рассмотрим, это Посредник.



Шаблон Медиатор используется для уменьшения сложности связи между несколькими объектами или классами.

Этот шаблон обеспечивает класс посредника Медиатор, который обрабатывает все коммуникации между различными классами и поддерживает простоту обслуживания кода с помощью слабой связи.

Диспетчерский центр аэропорта является примером шаблона Медиатор.

Центр следит за тем, кто может взлететь и приземлиться – и все коммуникации выполняются между самолетом и центром, а не между самолетами.

Эта идея центрального контроллера является одним из ключевых аспектов шаблона Медиатор.

На слайде интерфейс Медиатор определяет интерфейс для взаимодействия между объектами Colleague.

Класс ConcreteMediator реализует интерфейс Медиатор и координирует связь между объектами Colleague.

Он знает обо всех объектах Colleague и их целях в отношении взаимодействия.

Каждый объект ConcreteColleague взаимодействует с другими коллегами через посредника Медиатор.

Без этой схемы все Коллеги знали бы друг о друге, что приводило бы к сильному связыванию.

Но благодаря тому, что все коллеги общаются через одну центральный контроллер, у нас есть слабосвязанная система, но при этом мы сохраняем контроль над взаимодействием объектов.

Медиатор – это хороший выбор шаблона, когда связь между объектами сложная, но четко определена.

Когда между объектами вашего кода слишком много связей, нужно подумать об использовании этого шаблона.

```

public interface Mediator{
    public void send(String message, Colleague colleague);
}

public abstract Colleague{
    private Mediator mediator;
    public Colleague(Mediator m) {
        mediator = m;
    }
    //send a message via the mediator
    public void send(String message) {
        mediator.send(message, this);
    }
    //get access to the mediator
    public Mediator getMediator() {return mediator;}
    public abstract void receive(String message);
}

public class ConcreteColleague extends Colleague {
    public void receive(String message) {
        System.out.println("Colleague Received: " + message);
    }
}

```

```

public class ApplicationMediator implements Mediator{
    private ArrayList<Colleague> colleagues;
    public ApplicationMediator() {
        colleagues = new ArrayList<Colleague>();
    }
    public void addColleague(Colleague colleague) {
        colleagues.add(colleague);
    }
    public void send(String message, Colleague originator) {
        //let all other screens know that this screen has changed
        for(Colleague colleague: colleagues) {
            //don't tell ourselves
            if(colleague != originator) {
                colleague.receive(message);
            }
        }
    }
}

```

Для использования шаблона, сначала определим интерфейс Медиатор.

Хотя мы описали Colleague как интерфейс, более полезно использовать абстрактный класс.

Этот абстрактный класс хранит ссылку на объект Медиатор и определяет методы отправки и получения сообщений через Медиатор.

Далее мы реализуем интерфейс Медиатор, где определяем список коллег и определяем метод send, в котором один коллега посыпает сообщение другим коллегам.

Таким образом, клиентский код сначала создает объект ApplicationMediator.

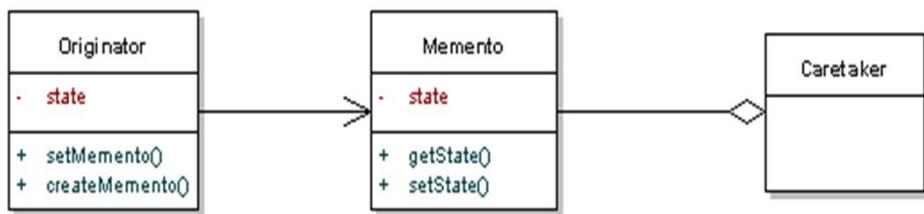
Затем создает объект ConcreteColleague, передавая в конструктор объект ApplicationMediator.

С помощью метода addColleague медиатора добавляем в список объекты ConcreteColleague.

И посыпаем сообщения методом send объекта ConcreteColleague.

Memento Pattern

Следующий шаблон, который мы рассмотрим, это Хранитель.



Шаблон Memento используется для восстановления состояния объекта в предыдущем состоянии.

В реальном мире, шаблон Хранитель используют как напоминание или ссылку о том, как что-то должно выглядеть.

Например, если вы решили разобрать некое устройство, чтобы заменить внутреннюю часть, у вас может быть такое же устройство, которое вы используете в качестве ссылки, чтобы вы могли вернуть свое устройство в исходное состояние.

На слайде, Оригинатор – это объект, который знает, как сохранить самого себя, это класс, который сохраняет свое состояние.

Объект Memento хранит информацию о состоянии Оригинатора.

Caretaker или Смотритель – это объект, который отслеживает все объекты Memento и не может их изменять.

Объект Caretaker отвечает за восстановление состояния объекта Оригинатор из Memento.

Шаблон Memento полезен, когда нужно обеспечить механизм отмены в приложении, когда внутреннее состояние объекта может потребоваться восстановить на более позднем этапе.

```

public class Memento {
    private String state;

    public Memento(String state){
        this.state = state;
    }

    public String getState(){
        return state;
    }
}

public class Originator {
    private String state;

    public void setState(String state){
        this.state = state;
    }

    public String getState(){
        return state;
    }

    public Memento saveStateToMemento(){
        return new Memento(state);
    }

    public void getStateFromMemento(Memento memento){
        state = memento.getState();
    }
}

```

Для использования шаблона Memento, мы сначала создаем класс Memento, который просто хранит состояние.

Затем в классе Originator мы используем Memento для сохранения и восстановления состояния.

Здесь у нас есть два метода get-set, которые просто работают с состоянием.

И есть два метода, которые сохраняют и извлекают состояние из Memento.

```

public class CareTaker {
    private List<Memento> mementoList = new
    ArrayList<Memento>();

    public void add(Memento state){
        mementoList.add(state);
    }

    public Memento get(int index){
        return mementoList.get(index);
    }
}

Originator originator = new Originator();
CareTaker careTaker = new CareTaker();

originator.setState("State #1");
careTaker.add(originator.saveStateToMemento());

originator.setState("State #2");
careTaker.add(originator.saveStateToMemento());

originator.getStateFromMemento(careTaker.get(0));
System.out.println("First saved State: " + originator.getState());

originator.getStateFromMemento(careTaker.get(1));
System.out.println("Second saved State: " + originator.getState());

```

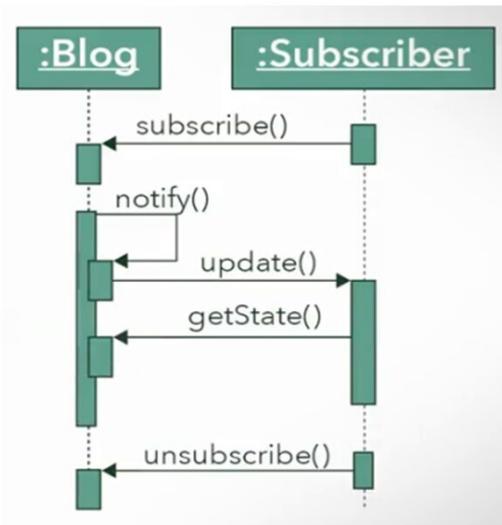
Затем у нас есть класс CareTaker, который хранит все Memento.

В клиентском коде мы устанавливаем состояние объекта Originator и сохраняем его в CareTaker, используя Memento.

Затем, когда нам нужно восстановить состояние, мы используем CareTaker, извлекая нужный нам Memento.

Observer Pattern

Следующий шаблон, который мы рассмотрим, это Observer или Наблюдатель, или Издатель-подписчик.



Представьте, что у вас есть любимый блог.

Каждый день вы посещаете его несколько раз, чтобы проверить новые записи в блоге.

Через некоторое время вам надоедает эта процедура.

И первое решение, которое вам приходит, – это написать скрипт для проверки блога на новые сообщения раз в секунду.

Но при дальнейшем рассмотрении вы поймете, что большинство сайтов не оценят этот шквал запросов и заблокируют ваш IP-адрес.

Чтобы этого избежать, вместо этого вы пишете скрипт для проверки блога на новые сообщения один раз в час.

Но к вашему разочарованию, это означает, что вы упускаете сообщения в блоге, которые предоставляют изменения в реальном времени.

Лучшим решением этой проблемы является то, что блог уведомляет вас каждый раз, когда добавляется новое сообщение.

Вы подписываетесь на блог, и каждый раз, когда будет опубликовано новое сообщение, блог будет уведомлять об этом каждого абонента, включая вас.

Таким образом, Subject, в нашем примере, блог, будет содержать список наблюдателей.

В этом примере мы можем думать об этих наблюдателях как о подписчиках блога.

И наблюдатели полагаются на блог, который сообщает им о любых изменениях в состоянии блога, например, при добавлении нового сообщения в блоге.

Поэтому, во-первых, у нас будет суперкласс Subject, который определяет два метода, которые позволяют новому наблюдателю подписаться, отписаться, и метод, уведомляющий всех наблюдателей о новом сообщении в блоге.

Этот суперкласс также будет иметь атрибут для отслеживания всех наблюдателей.

И у нас будет интерфейс наблюдателя с методом обновления.

Далее, класс Blog будет подклассом суперкласса Subject, а класс Subscriber будет реализовывать интерфейс наблюдателя.

Эти элементы дизайна необходимы для формирования отношений между объектом и наблюдателем.

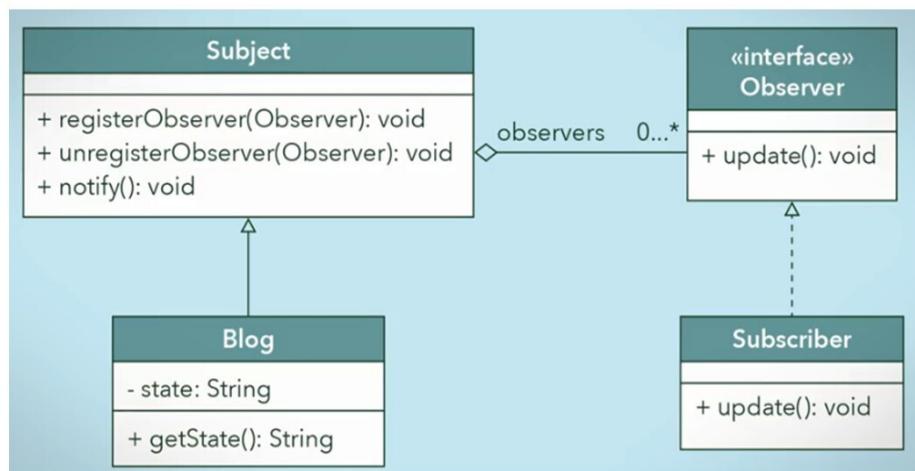
Чтобы сформировать отношения, подписчик должен подписаться на блог.

Далее, блог должен иметь возможность уведомлять своих подписчиков о том, что произошло изменение.

Этот метод вызывается только в том случае, если в блоге есть изменения, готовые для его подписчиков.

После того, как в блоге произойдут изменения, блог уведомит своих подписчиков через вызов метода update, а подписчики получат состояние блога через вызов метода getState.

Теперь, когда мы применили шаблон наблюдателя к конкретному примеру, давайте посмотрим, как эту идею можно абстрагировать в шаблон проектирования.



На этой диаграмме класса UML суперкласс имеет три метода: регистрировать наблюдателя, отменить регистрацию наблюдателя и уведомлять.

Подкласс класса Subject наследует эти методы, чтобы подписаться, отказаться от подписки и уведомить своих подписчиков.

Интерфейс наблюдателя имеет только метод обновления.

И класс Subscriber реализует интерфейс наблюдателя, предоставляя тело метода обновления.

Обратите внимание, что Subject может иметь ноль или более наблюдателей.

```
public class Subject {  
    private ArrayList<Observer> observers = new ArrayList<Observer>();  
  
    public void registerObserver(Observer observer) {  
        observers.add(observer);  
    }  
    public void unregisterObserver(Observer observer) {  
        observers.remove(observer);  
    }  
    public void notify() {  
        for (Observer o : observers) {  
            o.update();  
        }  
    }  
}
```

Теперь пришло время посмотреть на Java-код для шаблона Observer.
Здесь показано, как выглядит код для суперкласса.
Метод регистрации добавляет наблюдателя в список наблюдателей.
Другой метод удаляет наблюдателя из списка, а метод уведомления вызывает обновление
для каждого наблюдателя в списке.

```
public class Blog extends Subject {  
  
    private String state;  
  
    public String getState() {  
        return state;  
    }  
  
    // blog responsibilities  
    ...  
}
```

Далее, класс блога является подклассом Subject, наследуя его методы, добавляя метод
получения состояния.

```
public interface Observer {  
    public void update();  
}
```

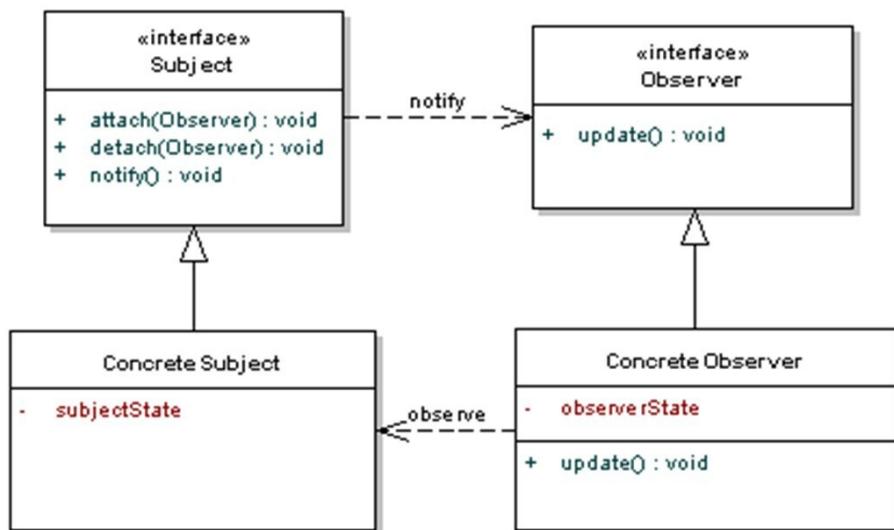
```
public class Subscriber implements  
Observer {  
  
    public void update() {  
        // get the blog change  
        ...  
    }  
}
```

Теперь, давайте посмотрим на интерфейс наблюдателя.

Интерфейс наблюдателя гарантирует, что все объекты наблюдателя ведут себя одинаково.

В классах наблюдателей необходимо реализовать только один метод – обновление.

Этот метод обновления вызывается «когда блог уведомляет подписчика об изменении».



Таким образом, шаблон Наблюдателя используется, когда между объектами существует связь один-ко многим, например, если один объект изменен, его зависимые объекты должны быть уведомлены автоматически.

В реальном мире, пример использования шаблона Наблюдатель, это подписка на новости.

Когда появляется новость, всем подписчикам приходит уведомление.

Идея шаблона простая – Наблюдатели хотят получать уведомления об изменении состояния Субъекта и регистрируют свой интерес к Субъекту с помощью присоединения к субъекту.

Когда что-то меняется в состоянии Субъекта, что может заинтересовать Наблюдателя, отправляется сообщение, которое вызывает метод обновления в каждом Наблюдателе.

Когда Наблюдатель больше не интересуется состоянием Субъекта, он может просто отсоединиться от Субъекта.

Это очень мощная связь – это означает, что любой объект может просто реализовать интерфейс Observer и получать обновления от Субъекта.

В общем, этот шаблон используется для слабого связывания.

Если есть объект, которому нужно поделиться своим состоянием с другими объектами, не зная, кто эти объекты, используется шаблон Наблюдатель.

Хотя некоторые шаблоны требуют, чтобы вы сами определяли интерфейсы шаблона, Observer – это случай, когда интерфейс уже содержится в пакете java.util.

```
import java.util.Observable;
public class DataStore extends Observable {
    private String data;
    public String getData() {
        return data;
    }
    public void setData(String data) {
        this.data = data;
        //mark the observable as changed
        setChanged();
    }
}
Screen screen = new Screen();
DataStore dataStore = new DataStore();
//register observer
dataStore.addObserver(screen);
//send a notification
dataStore.notifyObservers();

public class Screen implements Observer {
    @Override
    public void update(Observable o, Object arg) {
        //act on the update
    }
}
```

Пакет java.util содержит интерфейс Observer с методом update, который вызывается, если Субъект, расширяющий класс Observable пакета java.util, вызывает метод notifyObservers.

В этом примере, класс DataStore выступает как Субъект.

В методе setData мы вызываем метод setChanged класса Observable.

Это необходимо для того, чтобы пометить этот объект Observable как измененный, чтобы затем можно было уведомить наблюдателей о необходимости обновления.

Без этого вызова, объект Observable не отправит обновление своим наблюдателям.

Далее мы реализуем интерфейс Observer.

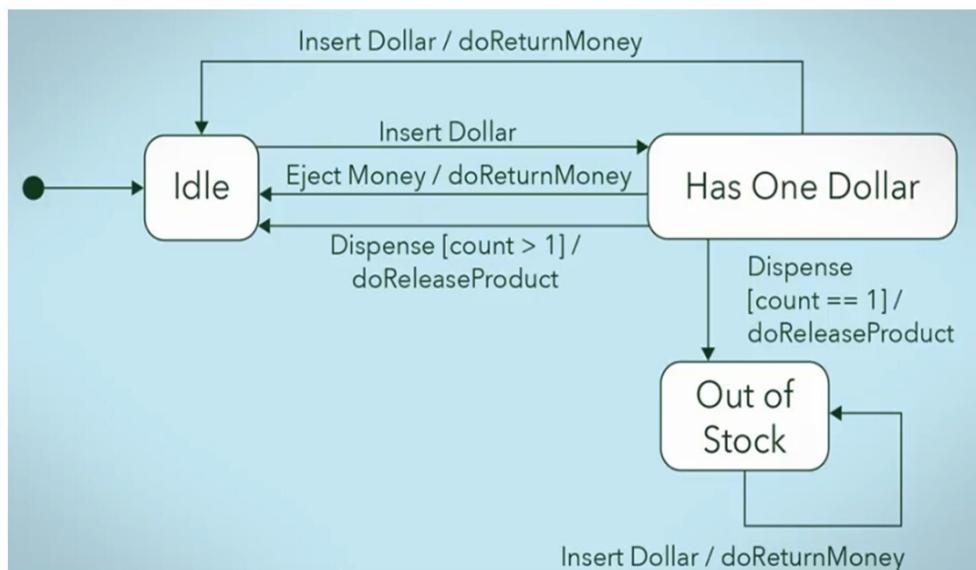
И в клиентском коде создаем объекты Observer и Observable.

Регистрируем объект Observer для объекта Observable.

И вызываем метод notifyObservers для запуска метода update наблюдателей.

State Pattern

Следующий шаблон, который мы рассмотрим, это Состояние.



Подумайте о том, что вы делаете в эту секунду.

Вы, очевидно, смотрите эту лекцию, но что еще вы делаете?

Вы сидите, или стоите, или лежите?

Скажем, я попросил вас потанцевать, но остаться в том же состоянии.

Если вы сидите, вы можете махать руками и плечами и качать головой.

Если вы стоите, вы сможете добавить движения ног.

Все, что я сделал, это сказал танцевать, и вы выбрали танец, который применим к нынешнему состоянию.

Мне не нужно было указывать, как должен выглядеть ваш танец.

Так как объекты в коде знают о своем текущем состоянии, вы можете использовать это в своем коде.

Объекты могут выбрать подходящее поведение, основанное на их текущем состоянии.

Когда изменяется их текущее состояние, это поведение может быть изменено.

В этом заключается шаблон состояния.

Когда вы должны использовать шаблон состояния?

Шаблон состояния в основном используется, когда вам нужно изменить поведение объекта на основе состояния, в котором объект находится во время выполнения.

Давайте посмотрим на пример.

Торговый автомат может быть представлен шаблоном состояния, так как он имеет несколько состояний и выполняет конкретные действия, основанные на этих состояниях.

Предположим, я захотел купить шоколадный батончик в торговом автомате.

Я подхожу к торговому автомату, вставляю монету и делаю свой выбор.

Затем машина выдает шоколадный батончик.

Я беру шоколадный батончик и наслаждаюсь им.

Это будет типичный сценарий, но давайте рассмотрим некоторые другие ситуации, которые могут произойти.

Что, если я подошел к торговому автомату, вставил монету, а потом решил, что я больше не хочу шоколадный батончик?

Тогда я нажму кнопку выброса денег, и машина вернет мне деньги.

Что, если в автомате закончились шоколадные батончики?

Торговый автомат должен отслеживать свое содержимое и уведомлять клиентов, когда больше не осталось определенного продукта.

На диаграмме состояний UML вы можете видеть, что есть три состояния.

И торговый автомат может находиться только в одном из этих состояний.

Вы также можете увидеть, что есть три триггера или события.

И существуют два действия, которые может сделать торговый автомат `doReturnMoney` и `doReleaseProduct`.

```
final class State { // singleton objects for states
    private State() {}  
  
    // all potential vending machine states as singletons
    public final static State Idle = new State();
    public final static State HasOneDollar = new State();
    public final static State OutOfStock = new State();
}
```

Давайте посмотрим, как это может выглядеть в коде.

Теперь, давайте подумаем о разных состояниях, в которых может находиться торговый автомат.

Сначала, машина находится в режиме ожидания, и ничего не происходит.

Когда вставляется монета, состояние торгового автомата меняется.

Он будет реагировать на запрос на возврат денег, возвращая деньги или отвечать на запрос о выдаче, выдавая продукт.

И третье состояние, в котором может находиться машина, это когда товар закончился, и машина не может торговаться.

```

public class VendingMachine {
    private State currentState;
    private int count;

    public VendingMachine( int count ) {
        if (count > 0) {
            currentState = State.Idle;
            this.count = count;
        } else {
            currentState = State.OutOfStock;
            this.count = 0;
        }
    }

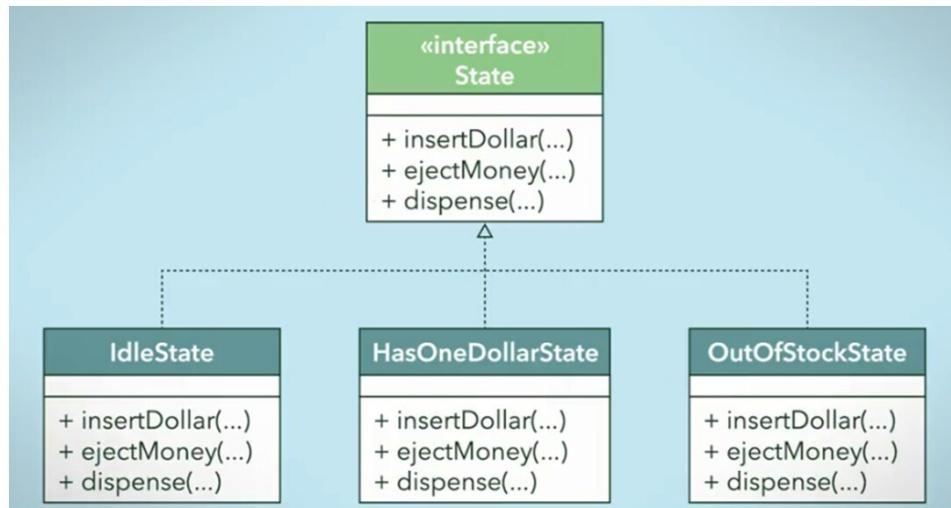
    // handle insert dollar trigger
    public void insertDollar() {
        if (currentState == State.Idle) {
            currentState = State.HasOneDollar;
        } else if (currentState == State.HasOneDollar) {
            doReturnMoney();
            currentState = State.Idle;
        } else if (currentState == State.OutOfStock) {
            doReturnMoney();
        }
    }
}

```

Давайте рассмотрим один из способов, которым мы можем представить это на Java.
Во-первых, мы создадим некоторые singleton объекты состояния.

Мы определяем наши три состояния как новые объекты состояния.

State.Idle будет обозначать состояние бездействия, state.HasOneDollar будет обозначать машину, в которую внесли деньги, и state.OutOfStock будет означать, что эта машина не торгует.



В нашем классе VendingMachine, currentState ссылается на конкретный объект состояния, в зависимости от текущего состояния торгового автомата.

Когда машина создается экземпляром с количеством товара больше нуля, текущее состояние устанавливается равным State.Idle.

Если количество товара не больше нуля, текущее состояние устанавливается равным State.OutOfStock.

Теперь, что произойдет, когда будет вставлена монета.

Вы можете видеть, что если текущее состояние торгового автомата было State.Idle, текущее состояние изменяется на State.HasOneDollar.

Если текущее состояние машины уже было State.HasOneDollar, мои деньги возвращаются, а текущее состояние становится – State.Idle.

Если текущее состояние машины State.OutOfStock, мои деньги возвращаются, и текущее состояние остается в State.OutOfStock.

```
public interface State {  
    public void insertDollar( VendingMachine vendingMachine );  
    public void ejectMoney( VendingMachine vendingMachine );  
    public void dispense( VendingMachine vendingMachine );  
}  
  
public class IdleState implements State {  
  
    public void insertDollar( VendingMachine vendingMachine ) {  
        System.out.println( "dollar inserted" );  
  
        vendingMachine.setState(  
            vendingMachine.getHasOneDollarState()  
        );  
    }  
  
    public void ejectMoney( VendingMachine vendingMachine ) {  
        System.out.println( "no money to return" );  
    }  
  
    public void dispense( VendingMachine vendingMachine ) {  
        System.out.println( "payment required" );  
    }  
}
```

Но эти объекты состояния не несут функциональности сами по себе.

Поэтому давайте теперь посмотрим, как мы можем правильно их перестроить, используя шаблон состояния.

Мы определим интерфейс состояния с методами, которыми должно реагировать состояние.

И у нас будут классы состояний, которые реализуют интерфейс состояния.

```

public class HasOneDollarState implements State {

    public void insertDollar( VendingMachine vendingMachine ) {
        System.out.println( "already have one dollar" );

        vendingMachine.doReturnMoney();
        vendingMachine.setState(
            vendingMachine.getIdleState()
        );
    }

    public void ejectMoney( VendingMachine vendingMachine ) {
        System.out.println( "returning money" );

        vendingMachine.doReturnMoney();
        vendingMachine.setState(
            vendingMachine.getIdleState()
        );
    }
}

```

Например, класс IdleState реализует интерфейс State.

Когда вставлена монета, вызывается метод insertDollar, который затем вызывает метод setState для объекта vendingMachine, который изменяет текущее состояние на HasOneDollarState.

```

public class VendingMachine {
    private State idleState;
    private State hasOneDollarState;
    private State outOfStockState;

    private State currentState;
    private int count;

    public VendingMachine( int count ) {
        // make the needed states
        idleState = new IdleState();
        hasOneDollarState = new HasOneDollarState();
        outOfStockState = new OutOfStockState();

        if (count > 0) {
            currentState = idleState;
            this.count = count;
        } else {
            currentState = outOfStockState;
            this.count = 0;
        }
    }

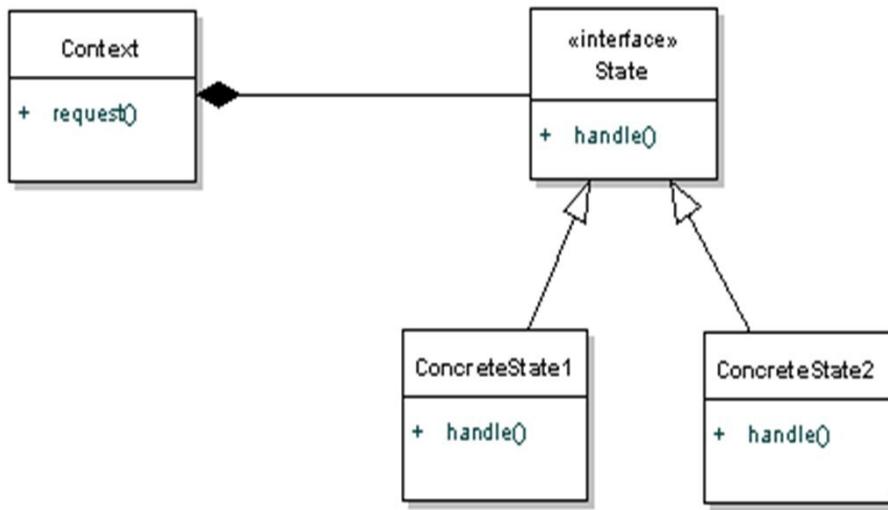
    public void insertDollar() {
        currentState.insertDollar( this );
    }

    public void ejectMoney() {
        currentState.ejectMoney( this );
    }

    public void dispense() {
        currentState.dispense( this );
    }
}

```

Аналогично, в классе HasOneDollarState, когда вызывается метод выталкивания денег, деньги возвращаются, и в торговом автомате вызывается setState, чтобы изменить состояние на состояние ожидания.



И конструктор класса vendingmachine будет создавать экземпляр каждого из классов состояний.

Текущее состояние будет ссылаться на один из этих объектов состояния.

Класс vendingmachine также будет иметь методы обработки событий, как и раньше, но теперь он делегирует обработку текущему объекту состояния.

```

public interface State {
    public void doAction(Context context);
}

public class StartState implements State {
    public void doAction(Context context) {
        System.out.println("Player is in start state");
        context.setState(this);
    }
    public String toString(){
        return "Start State";
    }
}

public class StopState implements State {
    public void doAction(Context context) {
        System.out.println("Player is in stop state");
        context.setState(this);
    }
    public String toString(){
        return "Stop State";
    }
}

public class Context {
    private State state;
    public Context(){
        state = null;
    }
    public void setState(State state){
        this.state = state;
    }
    public State getState(){
        return state;
    }
    public void action(){
        state.doAction(this);
    }
}
  
```

Таким образом, в шаблоне State поведение класса изменяется в зависимости от его состояния.

В шаблоне State мы создаем объекты, которые представляют различные состояния, и создаем объект контекста, поведение которого изменяется по мере изменения объекта состояния.

Контекст может иметь несколько внутренних состояний, каждый раз, когда вызывается метод request Контекста, передается сообщение в State для обработки.

Интерфейс State определяет общий интерфейс для всех конкретных состояний, инкапсулируя все поведение, связанное с определенным состоянием.

Когда объект Context изменяет состояние, тогда мы получаем другой объект ConcreteState, связанный с этим состоянием.

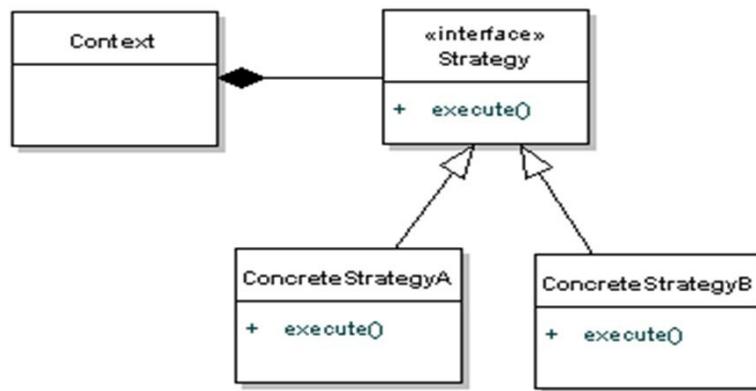
Этот шаблон используется, когда на поведение объекта должно влиять его состояние, и когда сложные условия связывают поведение объекта с его состоянием.

Здесь у нас есть интерфейс State и две его реализации.

И есть класс Context, поведение action которого изменяется, при изменении объекта State. Так как в этом методе вызывается метод конкретного состояния.

Strategy Pattern

Следующий шаблон, который мы рассмотрим, это Стратегия.



В шаблоне Стратегия поведение класса или его алгоритм можно изменить во время выполнения.

В шаблоне Стратегии мы создаем объекты, представляющие различные стратегии, и объект контекста, поведение которого изменяется в зависимости от его объекта стратегии.

Объект стратегии изменяет исполняющий алгоритм объекта контекста.

На слайде Контекст состоит из Стратегии.

Контекстом может быть все, что потребует изменения поведения – например, класс, который предоставляет функции сортировки.

Стратегия реализована как интерфейс, так что мы можем заменять объекты ConcreteStrategy без изменения Контекста.

```

public interface Strategy {
    public int doOperation(int num1, int num2);
}

public class OperationAdd implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 + num2;
    }
}

public class OperationSubtract implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 - num2;
    }
}

public class Context {
    private Strategy strategy;

    public Context(Strategy strategy){
        this.strategy = strategy;
    }

    public int executeStrategy(int num1, int num2){
        return strategy.doOperation(num1, num2);
    }
}

```

Context context = new Context(new OperationAdd());
context.executeStrategy(10, 5);
context = new Context(new OperationSubtract());
context.executeStrategy(10, 5);

Здесь у нас есть интерфейс стратегии и две его реализации.

Также есть класс Контекста, в конструктор которого передается объект стратегии.

И в зависимости от того, с каким объектом стратегии был создан объект контекста, поведение контекста меняется, согласно его методу executeStrategy.

Шаблон Стратегии используется там, где вы хотите выбрать определенный алгоритм во время выполнения.

Пример использования шаблона Стратегии, это сохранение файлов в разных форматах, выполнение различных алгоритмов сортировки.

Шаблоны Стратегия и Состояние похожи и отличие между ними следующее:

Состояния хранят ссылку на объект контекста, который их содержит. Стратегии нет.

Состояниям разрешено заменять себя (то есть изменить состояние объекта контекста на что-то другое), в то время как Стратегиям – нет. Объект контекста жестко привязан к объекту стратегии, в то время как состояние можно изменять динамически для одного и того же объекта контекста.

Стратегии передаются объекту контекста в качестве параметров, а состояния создаются самим объектом контекста.

Стратегии обрабатывают только одну конкретную задачу, тогда как состояния предоставляют базовую реализацию для всего, что объект контекста делает.

Template Pattern

Следующий паттерн, который мы рассмотрим, это Шаблон.

Допустим, вы являетесь шеф-поваром большой сети ресторанов.

Вы хотите, чтобы блюда были одинаковыми во всех ресторанах сети, поэтому вы даете инструкции для приготовления каждого блюда.

Ваши два самых популярных блюда – спагетти с томатным соусом и спагетти с курицей.

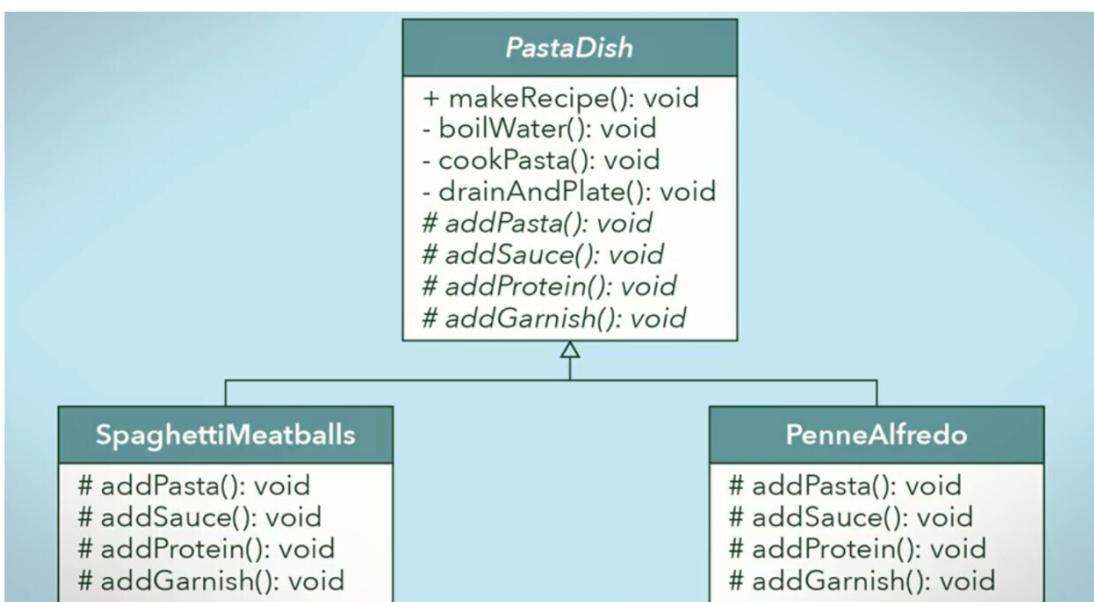
Оба блюда требуют, чтобы вы кипятили воду, готовили макароны, добавляли соус, добавляли другие составляющие и украшали тарелку.

Некоторые из этих шагов выполняются по-разному в зависимости от того, какое блюдо вы создаете.

Каждое блюдо имеет другой соус и гарнир.

Другие шаги будут одинаковыми для этих блюд.

И вы можете моделировать эту ситуацию с классом блюд с макаронами с помощью метода, который составляет рецепт для каждого подкласса этого суперкласса.



Метод знает общий набор шагов, чтобы приготовить блюдо.

Шаги, которые являются особыми для блюда, реализованы в его подклассе.

Это все элементы паттерна шаблонного метода.

Шаблонный метод определяет шаги алгоритма, откладывая реализацию некоторых шагов для подклассов.

Этот поведенческий паттерн связан с распределением обязанностей.

Паттерн шаблон лучше всего использовать, когда вы можете обобщить два класса в новый суперкласс.

После того, как вы используете обобщение, вы можете более эффективно повторно использовать объекты с использованием наследования, вы можете совместно использовать функциональность между классами и создать более четкий и понятный код.

Здесь у нас есть суперкласс `PastaDish` с шаблонным методом `makeRecipe`, который вызывает другие методы для шагов рецепта.

Некоторые шаги являются общими, тем не менее, некоторые шаги являются специальными для блюд, например, добавление соуса.

Таким образом, метод addSauce является абстрактным методом класса PastaDish, и он будет реализован в подклассах.

```
public abstract class PastaDish {  
    public final void makeRecipe() {  
        boilWater();  
        addPasta();  
        cookPasta();  
        drainAndPlate();  
        addSauce();  
        addProtein();  
        addGarnish();  
    }  
    protected abstract void addPasta();  
    protected abstract void addSauce();  
    protected abstract void addProtein();  
    protected abstract void addGarnish();  
  
    private void boilWater() {  
        System.out.println("Boiling water");  
    }  
    ..  
}
```

Здесь показан код суперкласса PastaDish, где общие методы реализованы и специфические методы объявлены как абстрактные.

Метод makeRecipe – это наш шаблонный метод.

Вы можете заметить, что этот метод отмечен как финальный.

В Java ключевое слово final означает, что объявленный метод не может быть переопределен подклассами.

Это означает, что ни один из подклассов не может иметь свою собственную версию makeRecipe.

Это обеспечивает согласованность шагов приготовления блюда и уменьшения избыточного кода.

```

public class SpaghettiMeatballs
extends PastaDish {
    protected void addPasta() {
        System.out.println("Add
spaghetti");
    }
    protected void addProtein() {
        System.out.println("Add
meatballs");
    }
    protected void addSauce() {
        System.out.println("Add
tomato sauce");
    }
    protected void addGarnish() {
        System.out.println("Add
Parmesan cheese");
    }
}

```



```

public class PenneAlfredo extends
PastaDish {
    protected void addPasta() {
        System.out.println("Add
penne");
    }
    protected void addProtein() {
        System.out.println("Add
chicken");
    }
    protected void addSauce() {
        System.out.println("Add
Alfredo sauce");
    }
    protected void addGarnish() {
        System.out.println("Add
parsley");
    }
}

```

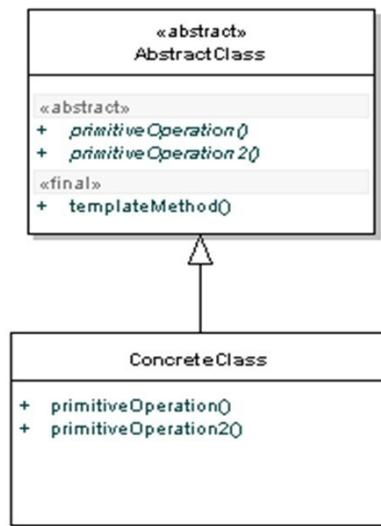
Оба подкласса расширяют класс `PastaDish`.

Каждый из них должен реализовать абстрактные методы, чтобы обеспечить подходящие ингредиенты для блюда.

Шаблонный метод наследуется подклассом и ведет себя так, чтобы приготовить блюдо с правильными ингредиентами.

Паттерн шаблона может быть полезен, если у вас есть два класса с аналогичной функциональностью.

Паттерн шаблона является практическим применением обобщения и наследования.



Таким образом, в паттерне Шаблон, абстрактный класс предоставляет определенные шаблоны для выполнения своих методов.

Его подклассы могут переопределять реализацию метода по мере необходимости, но вызов должен быть таким же, как был определен абстрактным классом.

Паттерн Шаблона используется, когда существуют две или более реализаций одного алгоритма.

В реальном мире шаблоны используются все время, например, для архитектурных планов.

Например, базовый план дома может иметь множество вариантов, таких как добавление веранды или использование другой системы отопления.

На слайде `AbstractClass` содержит метод `templateMethod`, который должен быть финальным, чтобы его нельзя было переопределить.

Этот метод шаблона использует другие методы алгоритма, но отделяется от фактической реализации этих методов.

Все методы, используемые этим методом шаблона, объявляются абстрактными, поэтому их реализация ложится на подклассы.

Класс `ConcreteClass` реализует все методы, используемые методом `templateMethod`, которые были определены как абстрактные в родительском классе.

И может быть много разных классов `ConcreteClass`.

Паттерн Шаблона использует принцип Голливуда: не звоните нам, мы позвоним вам.

Метод шаблона `templateMethod` в родительском классе управляет общим процессом, используя методы подкласса, когда это необходимо.

Существует четыре разных типа методов, используемых в родительском классе:

Это конкретные методы, которые являются стандартными методами, полезными для подклассов. Эти утилитные методы.

Также есть абстрактные методы. Это методы, не содержащие реализаций, которые должны быть реализованы в подклассах.

Есть Hook методы, содержащие реализацию по умолчанию, которые могут быть переопределены в некоторых классах. Hook методы предназначены для переопределения, конкретные методы – нет.

И есть методы шаблонов, которые вызывают любой из перечисленных выше методов, чтобы описать алгоритм без необходимости реализации деталей.

Паттерн Шаблон используется, когда поведение алгоритма может меняться, и вы позволяете подклассам реализовывать поведение с помощью переопределения.

И когда вы хотите избежать дублирования кода, реализуя разные варианты алгоритма в подклассах.

Обычный признак того, что вы должны использовать этот шаблон, – это когда вы обнаруживаете, что у вас есть два почти одинаковых класса, работающих с одной и той же логикой.

На этом этапе вы должны рассмотреть возможность использования шаблона для очистки вашего кода.

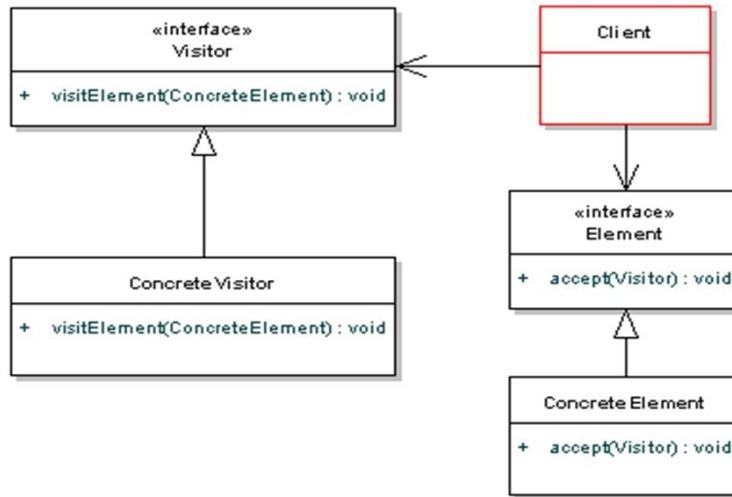
```
public class Football extends Game {  
  
    @Override  
    void endPlay() {  
        System.out.println("Football Game Finished!");  
    }  
  
    @Override  
    void initialize() {  
        System.out.println("Football Game Initialized! Start  
playing.");  
    }  
  
    @Override  
    void startPlay() {  
        System.out.println("Football Game Started. Enjoy the  
game!");  
    }  
}  
  
public abstract class Game {  
    abstract void initialize();  
    abstract void startPlay();  
    abstract void endPlay();  
  
    //template method  
    public final void play(){  
        //initialize the game  
        initialize();  
        //start game  
        startPlay();  
        //end game  
        endPlay();  
    }  
}
```

Здесь у нас есть абстрактный класс Game, шаблонный метод которого использует абстрактные методы, которые должны быть реализованы в подклассе.

Подкласс, реализовывая эти методы, определяет специфический алгоритм в рамках единой логики.

Visitor Pattern

Следующий паттерн, который мы рассмотрим, это Visitor или Посетитель.



В шаблоне Visitor мы используем класс Посетителя, который изменяет исполняющий алгоритм класса Element.

Таким образом, алгоритм выполнения Элемента может меняться в зависимости от изменения Посетителя.

В соответствии с шаблоном, объект-элемент должен принимать объект-посетитель, чтобы объект-посетитель обрабатывал операцию объекта-элемента.

Фактически шаблон Visitor создает внешний класс, который использует данные в других классах.

Шаблон Посетителя может предоставить дополнительную функциональность классу без его изменения.

Ядром этого шаблона является интерфейс Visitor.

Этот интерфейс определяет операцию посетителя для каждого типа ConcreteElement в структуре объекта.

Между тем, ConcreteVisitor реализует операции, определенные в интерфейсе Visitor.

ConcreteVisitor сохраняет локальное состояние, при обработке набора элементов.

Интерфейс Элемента просто определяет метод accept, чтобы позволить Посетителю выполнить какое-либо действие над этим элементом – и ConcreteElement реализует этот метод accept.

Шаблон используется, когда у вас есть различные и не связанные операции для выполнения со структурой объектов.

Это позволяет избежать добавления кода ко всей структуре объекта, и код создается отдельно, поэтому шаблон поощряет создание более чистого кода.

Вы можете запускать операции для набора объектов с разными интерфейсами.

Посетитель также используется, если вам нужно выполнить ряд несвязанных операций по классам.

Таким образом, если вы хотите отделить код логики от элементов, которые вы используете в качестве входных данных, можно использовать шаблон Посетитель.

```

//Element interface
public interface Visitable{
    public void accept(Visitor visitor);
}

public class Book implements Visitable{
    private double price;
    private double weight;
    //accept the visitor
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
    public double getPrice() {
        return price;
    }
    public double getWeight() {
        return weight;
    }
}

public interface Visitor{
    public void visit(Book book);
    //visit other concrete items
    public void visit(CD cd);
}

public class PostageVisitor implements Visitor {
    private double totalPostageForCart;
    //collect data about the book
    public void visit(Book book) {
        //assume we have a calculation here related to weight and price
        //free postage for a book over 10
        if(book.getPrice() < 10.0) {
            totalPostageForCart += book.getWeight() * 2;
        }
    }
    //add other visitors here
    public void visit(CD cd) {...}
    //return the internal state
    public double getTotalPostage(){
        return totalPostageForCart;
    }
}

```

Здесь у нас есть интерфейс элемента с методом, который принимает посетителя.

И есть реализация этого интерфейса и этого метода, в котором посетитель выполняет некую операцию над этим объектом элемента.

Таким образом, мы можем создать набор элементов и для каждого из этих элементов вызвать метод интерфейса, принимая в него объект посетителя.

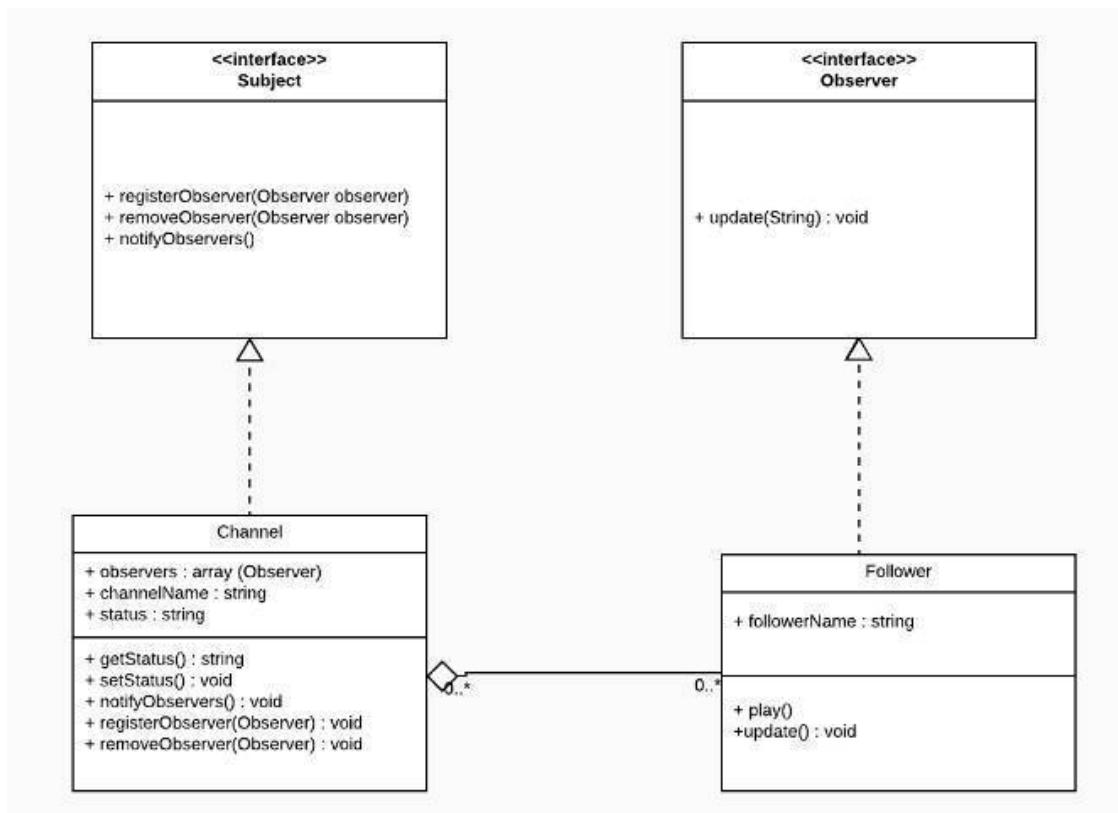
При этом будет вызван метод посетителя, который что-то сделает с данным объектом элемента, и при этом изменится общее состояние посетителя.

После обработки всего набора элементов, мы можем получить состояние посетителя и использовать его.

Здесь это метод getTotalPostage, который вернет поле totalPostageForCart, измененное в результате прохождения через набор элементов.

Задание

Youtube позволяет пользователям подписываться и следить за своими каналами. Это делается с использованием шаблона наблюдателя.
Заполните предоставленный код и используйте следующую диаграмму классов UML в качестве руководства.



Subject.java

```
public interface Subject {  
}
```

Channel.java

```
public class Channel implements Subject {  
}
```

Observer.java

```
public interface Observer {  
    public void update(String status);  
}
```

Follower.java

```
public class Follower implements Observer {  
}
```

Ответ

Subject.java

```
public interface Subject {  
    public void registerObserver(Observer observer);  
    public void removeObserver(Observer observer);  
}
```

```
    public void notifyObservers();
}
```

Channel.java

```
public class Channel implements Subject {
    private ArrayList<Observer> observers = new ArrayList<Observer>();
    private String channelName;
    String status;
    public Channel(String channelName, String status) {
        this.channelName = channelName;
        this.status = status;
    }
    public String getStatus() {
        return status;
    }
    public void setStatus(String status) {
        this.status = status;
        notifyObservers();
    }
    public void notifyObservers() {
        for (Observer obs : observers) {
            obs.update(this.status);
        }
    }
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
}
```

Observer.java

```
public interface Observer {
    public void update(String status);
}
```

Follower.java

```
public class Follower implements Observer {
    String followerName;
    public Follower(String followerName) {
        this.followerName = followerName;
    }
    public String getFollowerName() {
        return followerName;
    }
    public void setFollowerName(String followerName) {
        this.followerName = followerName;
    }
    public void update(String status) {
        //send message to followers that Channel is live.
    }
}
```

Вопросы

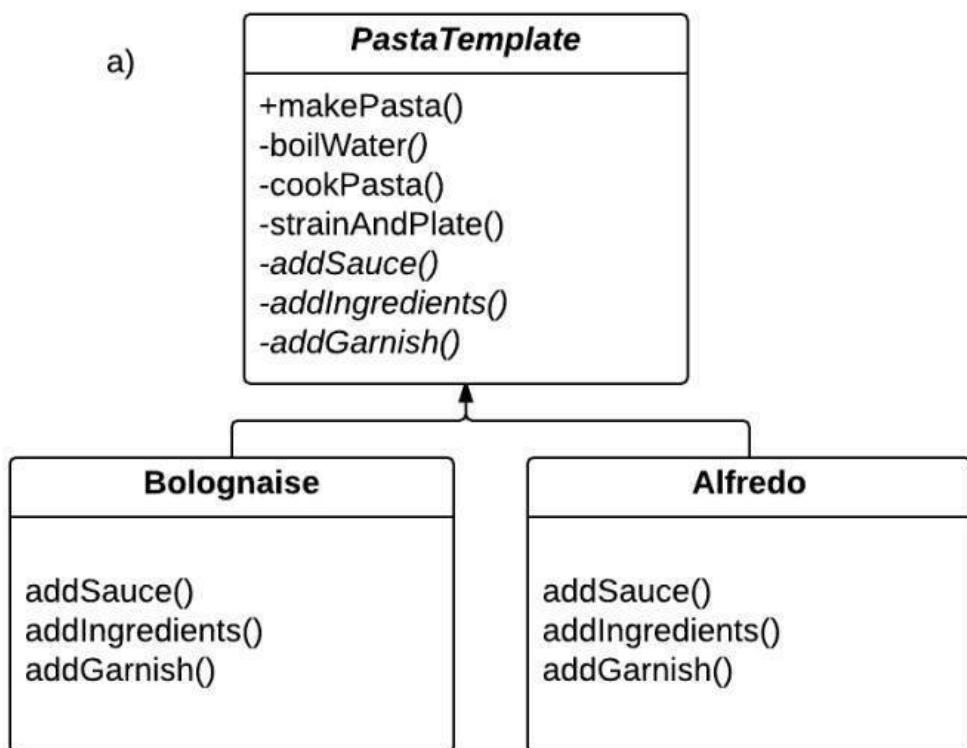
Вопрос 1

Выберите наиболее подходящий паттерн шаблона.

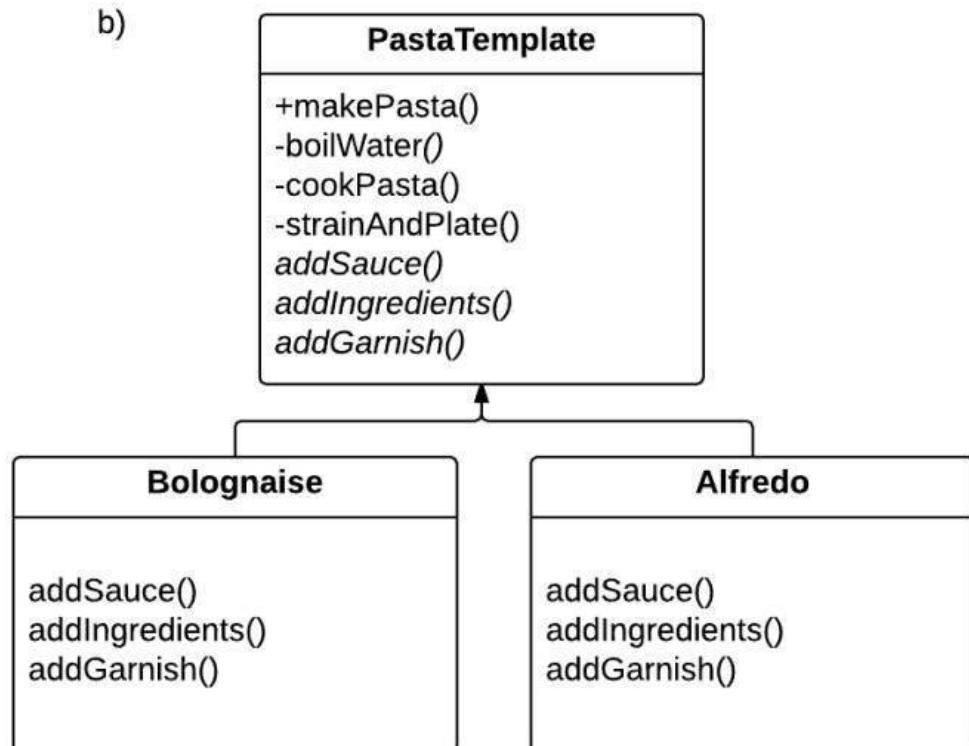
Приватный метод или переменная обозначается -, как -boilWater.

Метод, или класс, который является абстрактным, обозначается курсивом.

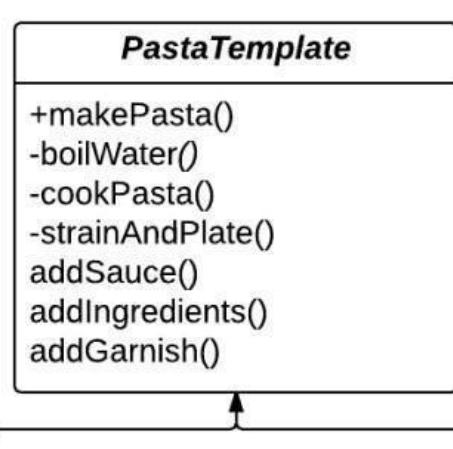
a)



b)



c)



Ответ d).

Вопрос 2

Какой является правильная ситуация для использования шаблона «Цепочка обязанностей»?

Вам нужен набор объектов, каждый из которых предоставляет информацию ответа на запрос.

У вас есть несколько потенциальных обработчиков, но только один будет иметь дело с запросом. +

Вам необходимо передать сообщение нескольким приемникам.

Вам необходимо делегировать набор задач иерархии объектов.

Вопрос 3

Какова цель инкапсуляции состояния в объект в шаблоне состояния?

Позволяет объекту текущего состояния решать, как достичь поведения, специфичного для состояния контекста. +

Удаляет многочисленные условия, которые трудно поддерживать. +

Превращает контекст в клиента состояния.

Позволяет копировать текущее состояние из одного экземпляра в другой.

Вопрос 4

Какие принципы проектирования использует командный шаблон?

Инкапсуляция, обобщение, скрытие информации.

Инкапсуляция, скрытие информации, слабое связывание.

Инкапсуляция, обобщение, слабое связывание. +

Обобщение, скрытие информации, слабое связывание.

Вопрос 5

Каковы минимальные требования шаблона Observer?

метод update в наблюдателях. +

переменная состояния, чтобы определить, были ли уведомлены наблюдатели.

методы добавления и удаления наблюдателей. +

метод уведомления наблюдателей. +

Вопрос 6

Когда вам, скорее всего, понадобится шаблон Mediator?

Если вы хотите разъединить класс, запрашивающий службу, от того, кто его предоставляет.

Когда ваш класс отправляет запрос, который может обрабатываться одним из нескольких обработчиков.

Когда у вас есть два класса с разными интерфейсами, которые вы должны соединить.

Когда вы координируете действия набора связанных классов. +

Вопрос 7

Вы кодируете часть программного обеспечения, которое следует аналогичной последовательности шагов. В зависимости от типа объекта эти шаги будут реализованы несколько иначе, но их порядок всегда один и тот же. Какой шаблон дизайна вы можете использовать?

Template pattern +

Mediator pattern

Command pattern

State pattern

Вопрос 8

Каковы важные роли в Command Pattern?

Sender, Receiver, Invoker

Command, Receiver, Invoker +

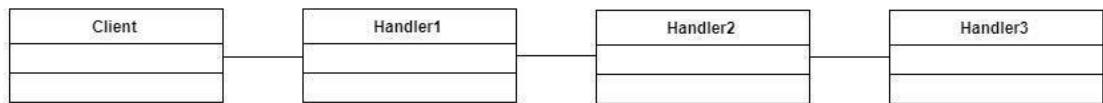
Command, Queue, Receiver

Delegate, Command, Requester

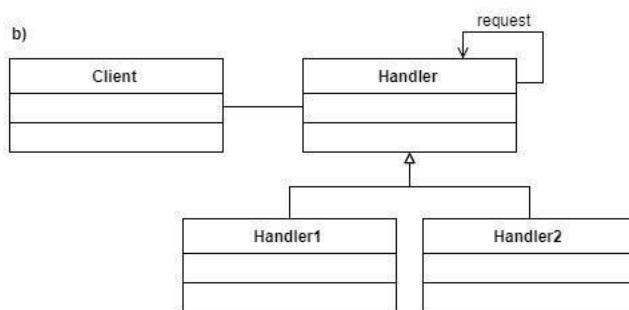
Вопрос 9

Выберите диаграмму классов UML шаблона «Цепочка обязанностей».

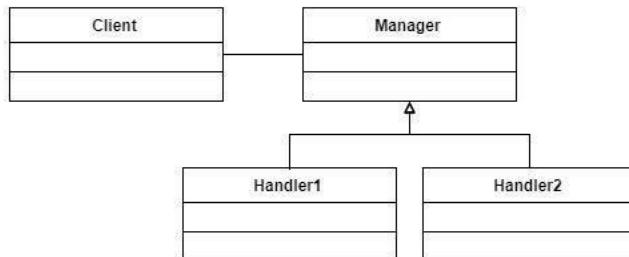
a)



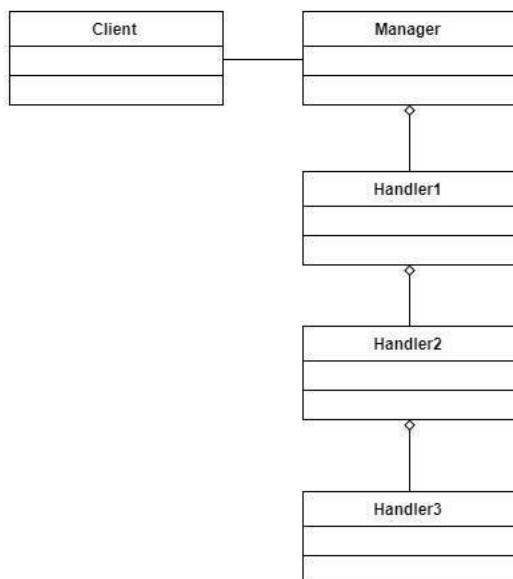
b)



c)



d)



Ответ b).

Вопрос 10

У вас есть машина, выполняющая сложную производственную задачу, с различными датчиками и различными компонентами машины, представленными разными классами.

Какой шаблон дизайна вы будете использовать для упорядочивания деталей?

Template

Command

Mediator +

Chain of Responsibility

Вопрос 11

У вас есть класс системы безопасности, и он имеет 3 режима: обычный, блокированный и открытый. Какой шаблон вы бы использовали для моделирования поведения в этих разных режимах?

Observer

Template

State +

Mediator

Вопрос 12

Один из ваших классов представляет собой почтовый ящик, а другой – владельца почтового ящика. Человек хотел бы знать, когда придет новая почта. Какой шаблон дизайна вы, вероятно, используете?

Observer +

Mediator

State

Command

MVC Pattern

Давайте рассмотрим небольшой продуктовый магазин.



Кассиры этого магазина жалуются на старую кассовую систему, где им приходится вручную печатать цену каждого товара.

В то же время клиенты устают ждать в очереди в кассу.

Таким образом, им нужна новая система для ввода заказов.

Другими словами, пользовательский интерфейс.

Клиент и кассир должны увидеть товары, которые вводятся в заказ с помощью сканера штрих-кода.

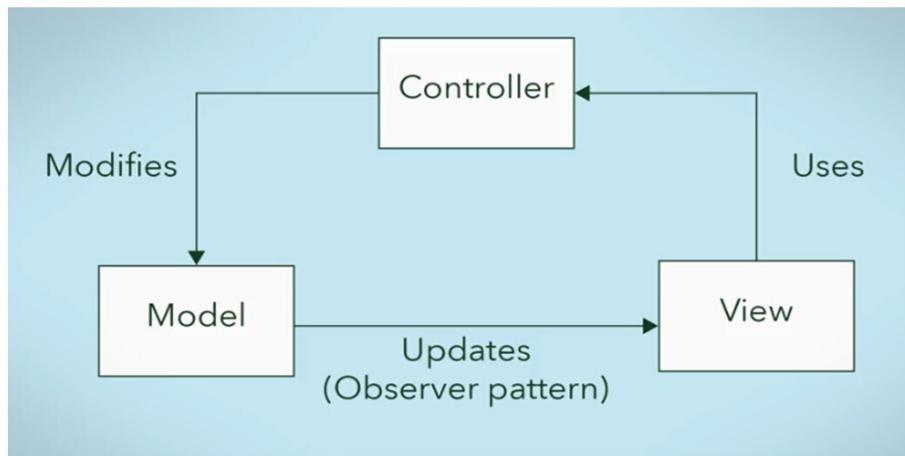
Они также должны иметь возможность видеть общую сумму счета.

Кассирам это понравится, потому что им больше не нужно будет вводить цены.

И клиентам тоже понравится, потому что чеки будут обрабатываться быстрее, и они смогут увидеть цены товара на экране.

Если будут допущены ошибки, кассир сможет легко внести исправления, и все будут довольны.

Каждый раз, когда вы слышите о пользовательском интерфейсе, вам следует использовать шаблон MVC.



MVC – это модель-вид-контроллер.

Шаблон MVC разделяет обязанности системы, предлагающей пользовательский интерфейс на эти три части.

Здесь показана диаграмма простого шаблона MVC.

Во-первых, давайте поговорим о модели.

Модель содержит низлежащие данные и логику, которые пользователи хотят видеть и использовать.

Ключевой особенностью шаблона MVC является то, что модель является автономной.

Модель имеет все состояние, методы и другие данные, которые необходимы, чтобы модель существовала сама по себе.

Следующий компонент паттерна – это вид View.

Вид View дает пользователю возможность увидеть модель или, по крайней мере, ее части.

Иногда в представлении View также будут элементы взаимодействия, такие как кнопки и поля, которые позволяют пользователю взаимодействовать с системой.

Модель – это back end, а вид – это front end, это уровень представления.

Хотя здесь показан только один вид, может быть несколько видов, и все они используются с одной и той же моделью.

Когда какое-то значение изменяется в модели, об этом должно сообщаться виду для его обновления.

Это делается с использованием шаблона проектирования Observer.

Помните, что в шаблоне наблюдателя, наблюдатели действуют как подписчики.

В этом случае любое представление View также является наблюдателем.

Когда модель изменяется, она уведомляет об этом все виды, которые подписаны на нее.

Вид может также иметь способы для пользователя внести изменения в данные модели.

В шаблоне MVC вид напрямую не отправляет запросы модели.

Вместо этого информация о пользовательском взаимодействии передается контроллеру, который отвечает за интерпретацию этих запросов и изменение модели.

Таким образом, представление View отвечает только за внешний вид системы, а модель фокусируется исключительно на управлении информацией для системы.

И таким образом, шаблон MVC использует принцип проектирования – разделение обязанностей.

В общем, модель соответствует объектам сущности, полученным из анализа пространства задачи для системы.

Вид соответствует граничному объекту на краю вашей системы, имея дело с пользователями.

Контроллер соответствует объекту контроля, который принимает события и координирует действия.

Как этот шаблон может быть реализован?

Начнем с самой важной части, модели.

```

import java.util.*;

public class StoreOrder extends Observable {
    private ArrayList<String> itemList;
    private ArrayList<BigDecimal> priceList;

    public StoreOrder() {
        itemList = new ArrayList<String>();
        priceList = new ArrayList<BigDecimal>();
    }

    public String getItem( int itemNum ) {
        return itemList.get(itemNum);
    }

    public String getPrice( int itemNum ) {
        return priceList.get(itemNum);
    }

    public ListIterator<String> getItemList() {
        ListIterator<String> itemItr =
            itemList.listIterator();
        return itemItr;
    }

    public ListIterator<BigDecimal>
    getPriceList() {
        ListIterator<String> priceItr =
            priceList.listIterator();
    }
}

return priceItr;
}

public void deleteItem( int itemNum ) {
    itemList.remove(itemNum);
    priceList.remove(itemNum);
    setChanged();
    notifyObservers();
}

public void addItem( int barcode ) {
    // code to add item (probably used with
    // a scanner)
    // prices are looked up from a database
    ...
    setChanged();
    notifyObservers();
}

public void changePrice( int itemNum,
    BigDecimal newPrice ) {
    priceList.set(itemNum,newPrice);
    setChanged();
    notifyObservers();
}
}

```

Модель должна существовать сама по себе без видов или контроллеров.

Конечно, вы, вероятно, будете использовать виды и контроллеры для просмотра и изменения модели.

Но модель не должна зависеть от вида или контроллера, чтобы существовать.

Поскольку наш вид будет наблюдателем, мы должны сделать модель, расширяющую класс Observable, чтобы модель, в нашем случае класс заказа, позволяла добавлять виды в качестве наблюдателей.

Таким образом, они будут действовать как подписчики класса заказа товара, и обновляться всякий раз, когда заказ обновляется.

Обратите внимание: наша модель не имеет каких-либо элементов интерфейса и не знает о каких-либо представлениях.

Она обновляет представления через шаблон наблюдателя.

Всякий раз, когда происходят изменения, метод setChanged указывает, что произошло изменение, а метод notifyObservers уведомляет представление, чтобы оно могло себя обновить.

```

import java.util.*;
import javax.swing.JFrame;
// ...etc.

public class OrderView extends JPanel implements
Observer, ActionListener {
    // Controller
    private OrderController controller;

    // User-Interface Elements
    private JFrame frame;
    private JButton changePriceButton;
    private JButton deleteItemButton;
    private JTextField newPriceField;
    private JLabel totalLabel;
    private JTable groceryList;

    private void createUI() {
        // Initialize UI elements. e.g.:
        deleteItemButton = new JButton("Delete
        Item");
        add(deleteItemButton);
        ...
        // Add listeners. e.g.:
        deleteItemButton.addActionListener(this);
        ...
    }

    public void update ( Observable s, Object arg ) {
        display(((StoreOrder) s).getItemList(),
        ((StoreOrder) s).getPriceList());
    }

    public OrderView(OrderController controller) {
        this.controller = controller;
        createUI();
    }

    public void display ( ArrayList itemList,
ArrayList priceList ) {
        // code to display order
        ...
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == deleteItemButton) {
            controller.deleteItem(groceryList,
            getSelectedRow());
        }
        else if (event.getSource() ==
        changePriceButton) {
            BigDecimal newPrice = new
            BigDecimal(newPriceField.getText());
            controller.changePrice
            (groceryList.getSelectedRow(),
            newPrice);
        }
    }
}

```

Здесь показано представление.

Класс OrderView реализует интерфейс наблюдателя, поэтому он определяет метод update.

Что происходит, когда мы используем представление для изменения модели?

Посмотрите здесь на обработку событий, и обратите внимание, что здесь нет вызовов методов модели.

Вместо этого вызываются методы контроллера.

```

public class OrderController {

    private StoreOrder storeOrder;
    private OrderView orderView;

    public OrderController(StoreOrder storeOrder, OrderView orderView) {
        this.storeOrder = storeOrder;
        this.orderView = orderView;
    }

    public void deleteItem( int itemNum ) {
        storeOrder.deleteItem( itemNum );
    }

    public void changePrice( int itemNum, BigDecimal newPrice ) {
        storeOrder.changePrice( itemNum, newPrice );
    }
}

```

В этом примере контроллер очень простой.

Контроллер должен иметь ссылки как на вид, так и на модель, с которой он соединяется.

И контроллер не вносит изменения в состояние модели напрямую.

Он вызывает методы модели для внесения изменений.

Используя контроллер, вы делаете код лучше несколькими способами.

Прежде всего, вид может сфокусироваться на своей главной цели, представлять пользовательский интерфейс.

Контроллер берет на себя ответственность за интерпретацию ввода от пользователя и работу с моделью на основе этого ввода.

Разделение этих задач делает код более чистым и более простым в изменении.

Также, вставляя контроллер между моделью и вид, вид больше не связан тесно с моделью.

При разработке, вы можете добавлять функции к модели и тестировать их задолго до того, как использовать их в представлении.

Определяющей особенностью шаблона MVC является разделение задач между бэкендом, фронтэндом и координацией между ними.

Задание

Компания попросила вас создать систему, позволяющую менеджерам просматривать, редактировать и добавлять информацию о сотрудниках.

Вы решили создать им веб-приложение.

Они сообщили вам, что они могут принять решение о дальнейшем развитии и расширении и хотели бы, чтобы система была гибкой для расширения.

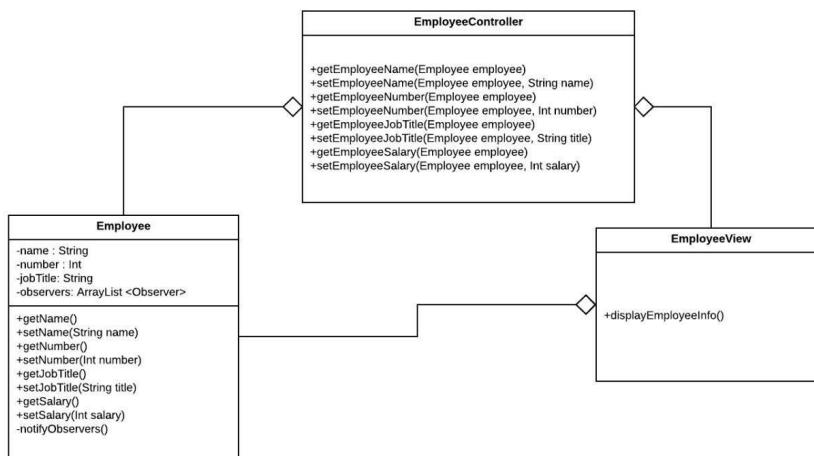
В этом случае следует использовать шаблон MVC.

Создайте диаграмму классов UML, которая отображает базовый шаблон MVC для этого веб-приложения.

Система должна отслеживать имя сотрудника, идентификационный номер, должность и заработную плату.

Контроллер должен иметь возможность получать свойства модели сотрудника (методы getter) и изменять свойства (методы setter). В представлении должна отображаться только информация о сотруднике.

Ответ:



Принципы проектирования. Принцип подстановки Лисков

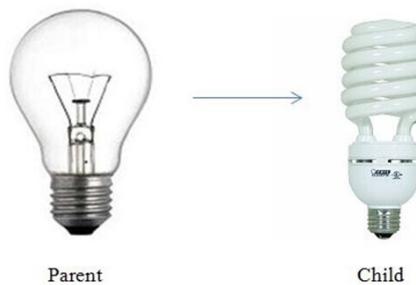
В объектно-ориентированном программировании разработчики могут определять свои структуры данных и использовать их для инкапсуляции набора атрибутов и методов.

Эти структуры данных известны как «классы». Они позволяют разработчикам создавать абстрактные представления объектов реального мира или теоретические концепции в программной системе.

В общем, объекты и идеи не существуют изолированно, а это значит, что они связаны, состоят из частей или связаны с чем-то другим.

Как эти взаимосвязи реализуются в объектно-ориентированном программировании?

Например, с помощью такой объектно-ориентированной конструкции, как наследование.



Наследование – это концепция, которая позволяет классу быть «наделенным» характеристиками и поведением другого класса.

В этой конструкции унаследованный класс известен как «базовый класс», а наследующий класс является «подклассом».

Наследование является основой объектно-ориентированного программирования, поскольку позволяет подклассам получать те же атрибуты и методы, что и базовый класс.

Кроме того, наследующие классы могут добавлять свои собственные характеристики и поведение, позволяя им стать более «специализированными».

Наследование позволяет подклассу стать полиморфным, поскольку наследование позволяет подклассу стать подтипом базового класса.

Подтипованием также позволяет использовать подклассы в качестве замены или подстановки их базового класса.

Подстановка гласит, что:

Любой класс, S, может быть использован для замены класса В, тогда и только тогда, когда S является подтиповом В.

Проще говоря, любой подкласс может работать за базовый класс.

Из-за наследования ожидается, что подкласс будет иметь те же характеристики и вести себя также.

Хотя это легко понять, это создает определенные трудности при разработке объектно-ориентированной системы.

Разработчики должны определять, когда целесообразно использовать наследование.

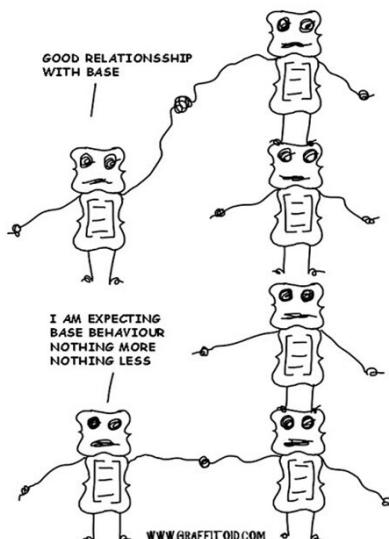
Неправильное применение наследования может привести к тому, что классы будут вести себя нежелательным образом.

Базовый класс – это наиболее обобщенный класс, и поэтому его атрибуты и поведение должны быть обобщением.

Его атрибуты и методы должны быть достаточно широкими, чтобы все подклассы могли их использовать.

Если наследование не используется правильно, это может привести к нарушению «Принципа подстановки Лисков».

LISKOV SUBSTITUTION PRINCIPLE



Этот принцип использует подстановку, чтобы определить, правильно ли использовалось наследование.

В принципе подстановки Лисков говорится, что:

Если класс, S, является подтипов класса B, тогда S можно использовать для замены всех экземпляров B без изменения поведения программы.

Логика этого проста.

Если S является подтипов B, тогда можно ожидать, что S будет иметь такое же поведение, как B.

Следовательно, S можно использовать вместо B, и это не повлияет на программное обеспечение.

Это означает, что наследование может быть проверено путем применения замены.

Существует ряд ограничений, которые Принцип подстановки Лисков накладывает на подклассы для обеспечения надлежащего использования наследования.

Условие, используемое для определения того, должен или не должен базовый класс вызывать метод, не может быть «усилено» подклассом.

Состояние программы после выполнения метода не может быть «ослаблено» подклассом.

Инвариантные условия, существующие в базовом классе, также должны оставаться инвариантными в подклассе.

Неизменяемые характеристики базового класса не должны изменяться подклассом.

Условие, используемое для определения того, должен или не должен базовый класс вызывать метод, не может быть «усилен» подклассом.

То есть подкласс не может добавлять дополнительные условия для определения того, следует ли вызывать метод.

Состояние программы после выполнения метода не может быть «ослаблено» подклассом.

Это означает, что подкласс должен обеспечивать состояние программы в том же состоянии, что и базовый класс после вызова метода.

Подклассам разрешено «укреплять» пост-условие программы.

Например, если базовый класс устанавливает будильник на определенную дату, подкласс должен делать то же самое, но результат может быть более точным, если подкласс еще и задаст также определенный час.

Инвариантные условия, существующие в базовом классе, также должны оставаться инвариантными в подклассе.

Так как ожидается, что инвариантные условия будут неизменными, подкласс не должен изменять их, поскольку это может вызвать побочный эффект в поведении базового класса или программы.

Неизменяемые характеристики базового класса не должны изменяться подклассом.

Поскольку классы могут изменять свои собственные характеристики, подкласс может изменять все характеристики, которые он наследует от базового класса.

Однако базовый класс может инкапсулировать атрибуты, которые должны быть фиксированными значениями.

Эти значения можно определить, наблюдая, изменились ли они в программе в базовом классе.

Если они не изменились, то эти атрибуты считаются неизменяемыми.

Подклассы могут обойти эту проблему, объявив и изменив свои собственные атрибуты.

Атрибуты подкласса не видны базовому классу и поэтому не влияют на поведение базового класса.

Эти правила не обеспечиваются программно объектно-ориентированным языком.

Их нужно соблюдать вручную.

Переопределение поведения базового класса может дать преимущества.

Подклассы могут улучшить поведение базового класса, не изменяя ожидаемых результатов этого поведения.

Как пример, давайте посмотрим на класс, являющийся абстракцией универмага.

Базовый класс может реализовать самый простой алгоритм поиска, который просто выполняет итерацию по всему списку товаров, которые продаёт магазин.

Подкласс может переопределить этот метод и обеспечить лучший алгоритм поиска.

Хотя подход, используемый подклассом для поиска, отличается, ожидаемое поведение и результат одинаковы.

Принцип замены Лискова помогает нам определить, правильно ли использовалось наследование.

Если ожидаемое поведение при использовании подкласса и базового класса отличается, тогда этот принцип был нарушен.

Открыто-закрытый принцип

Как разработчик программного обеспечения, вы должны стремиться создавать системы, которые являются гибкими и многоразовыми.

Гибкие код позволяет упростить расширение системы.

Многоразовый код означает, что вам не нужно переопределять то, что уже было сделано. Достижение этих целей поможет сделать ваш код более удобным.

Шаблоны проектирования, – это всевозможные методы, которые вы можете использовать, чтобы написать гибкий и многоразовый код.

Это средство, с помощью которого вы можете улучшить качество своего кода.

Все шаблоны проектирования соответствуют базовому набору принципов проектирования, в которых рассматриваются такие вопросы, как гибкость и возможность повторного использования.

Один из этих принципов называется принципом «открытый/закрытый».



Этот принцип гласит, что классы должны быть открыты для расширения, но закрыты для изменения.

Теперь, что это значит?

Как следует из названия этого принципа, есть две части, открытые и закрытые.

Давайте рассмотрим, что означает каждая часть.

Вы должны рассматривать класс как «закрытый» для редактирования, как только он был протестирован для правильной работы.

Класс должен вести себя так, как вы ожидаете, что он будет себя вести.

Все атрибуты и поведение инкапсулированы и оказались стабильными в вашей системе.

Класс или любой экземпляр класса не должен останавливать работу вашей системы или навредить ей.

Закрытая часть принципа не означает, что вы не можете вернуться в класс, чтобы внести изменения в него во время разработки.

Ожидается, что изменения возможны на этапе цикла разработки – проектирование и анализ.

На этом этапе ожидаются изменения классов.

После того, как вы уже завершили большинство своих проектных решений и внедрили большую часть своей системы, вам следует рассмотреть возможность закрытия ваших классов.

Во время работы вашего программного обеспечения определенные классы должны быть закрыты, чтобы избежать появления нежелательных побочных эффектов.

Конечно, вы должны по-прежнему исправлять закрытый класс, если есть какие-либо ошибки или неожиданные действия.

Итак, что вы будете делать, если вам нужно добавить дополнительные функции?

Как вы расширите свою систему?

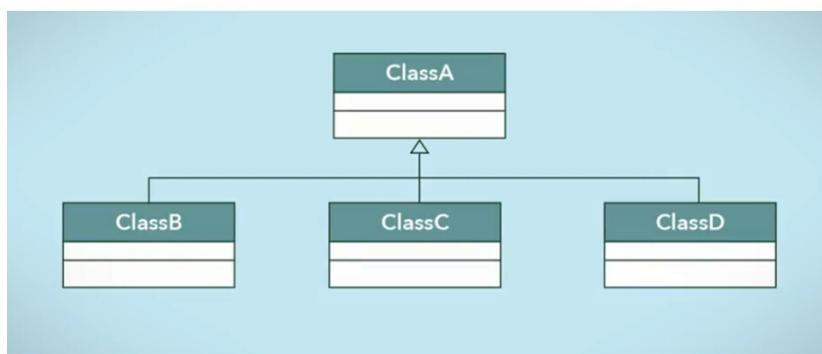
Здесь начинает работать открытая часть принципа дизайна.

Существует два разных взгляда на открытую часть принципа.

Но оба они считают, что класс должен быть открытым, если вы все еще строите систему с его использованием.

Существует два разных способа расширения вашей системы с использованием принципа открытости.

Первый способ – через наследование суперкласса.



Идея состоит в том, что, если вы хотите добавить больше атрибутов и поведения в класс, который считается закрытым, вы можете просто использовать наследование для его расширения.

Таким образом, ваши подклассы будут по-прежнему иметь все оригинальные функции суперкласса.

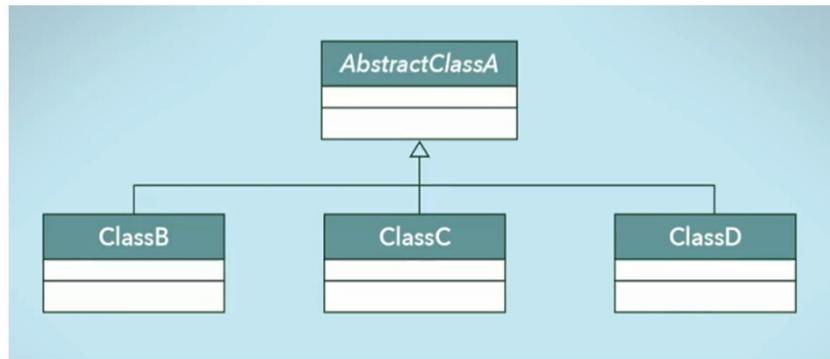
Но теперь вы можете добавить дополнительные функции в подклассы.

Это помогает сохранить целостность суперкласса.

И помните, что подклассы также могут быть расширены, поэтому вы можете использовать принцип открытый закрытый, чтобы постоянно расширять вашу систему столько, сколько вы хотите.

Вы можете достичь точки, когда вы больше не хотите, чтобы класс расширялся, и в этом случае вы можете объявить класс финальным, что предотвратит дальнейшее наследование.

Ключевое слово «final» также можно использовать для методов.



Второй способ, с помощью которого класс можно считать открытым для расширения, это является ли класс абстрактным и обеспечивает ли он принцип открытый закрытый посредством полиморфизма.

Абстрактный класс может объявлять абстрактные методы только с помощью сигнатур метода.

Каждый конкретный подкласс должен обеспечивать собственную реализацию этих методов.

И вы можете расширять свою систему, предоставляя различные реализации для каждого метода.

Вы также можете использовать интерфейс для полиморфизма.

Но имейте в виду, что с интерфейсом вы не сможете определить общий набор атрибутов.

Принцип открытый закрытый используется для того, чтобы поддерживать стабильные части вашей системы отдельно от других частей.

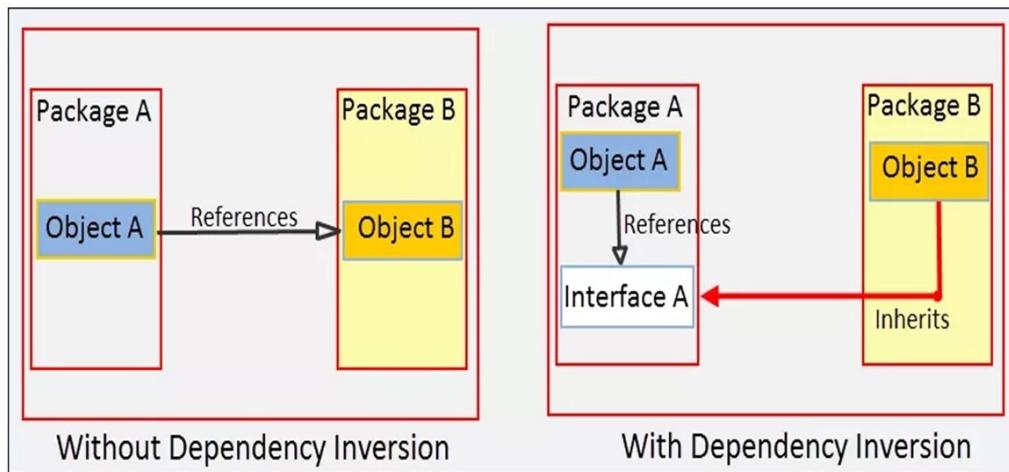
Используя расширение вместо изменения, вы можете работать с различными частями, не вводя нежелательные побочные эффекты в стабильные части.

Разные части системы должны быть изолированы друг от друга.

Принцип открытый закрытый – это концепция, которая помогает поддерживать стабильность системы, закрывая классы для изменений и позволяя системе открываться для расширения посредством использования наследования или интерфейсов.

Инверсии зависимостей

Общая проблема, которую вам нужно будет учитывать при разработке ваших систем, является зависимость.



Зависимость программного обеспечения, по сути, определяет, как связаны разные компоненты вашего программного обеспечения.

Если части вашей системы сильно связаны, то степень, в которой они полагаются друг на друга, считается высокой, в то время как низкая зависимость означает, что ваша система имеет более низкую степень связывания.

Зависимость является важной темой, так как она определяет, как легко вы можете вносить изменения в свою систему.

В реальном мире наши тела зависят от воды для выживания.

Мы не можем заменить воду хлебом и получить тот же результат, как если бы мы пили воду.

В конечном итоге вы столкнетесь с той же проблемой в программном обеспечении.

Попытка заменить один класс или ресурс на что-то другое может быть нелегкой, или иногда даже невозможной.

Это может быть связано с тем, что ваша система зависит от конкретной функции, которая может быть предложена только этим конкретным классом.

Чтобы решить эту проблему, мы используем принцип инверсии зависимостей, который помогает сделать системы более надежными и гибкими.

Принцип гласит, что модули высокого уровня должны зависеть от обобщений высокого уровня, а не от деталей низкого уровня.

Это означает, что ваши клиентские классы должны зависеть от интерфейса или абстрактного класса вместо ссылки на конкретные ресурсы и что ваши конкретные ресурсы должны иметь свое поведение, обобщенное в интерфейс или абстрактном классе.

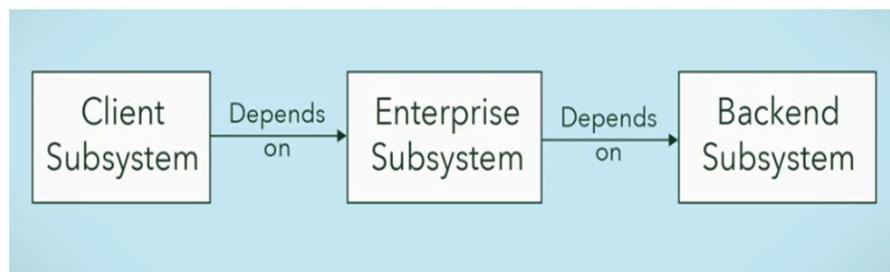
Идея состоит в том, что интерфейсы и абстрактные классы считаются ресурсами высокого уровня, а конкретный класс считается ресурсом низкого уровня.

Интерфейс или абстрактный класс определяет общий набор поведений, а конкретные классы обеспечивают реализацию этих действий.

Таким образом, ваши клиентские классы могут быть независимы от низкоуровневой функциональности.

Все шаблоны проектирования, которые мы рассматривали, основаны на этом принципе.

Давайте рассмотрим разницу между зависимостью низкого уровня и зависимостью высокого уровня, чтобы продемонстрировать, как работает принцип инверсии зависимостей.



В стандартном программном обеспечении ваша архитектура системы может выглядеть как показано на слайде.

Ваши подсистемы напрямую зависят друг от друга, что означает, что ваш клиентский класс будет напрямую ссылаться на конкретный класс в вашей подсистеме уровня предприятия, который будет напрямую ссылаться на конкретный класс в вашем бэкэнде.

Эта форма зависимости будет зависимостью низкого уровня, потому что клиентские классы ссылаются на конкретный класс.

```
public class ClientSubsystem {  
    public QuickSorting enterpriseSorting;  
    /* Constructors and other attributes go here */  
  
    public void sortInput(List customerList) {  
        this.enterpriseSorting.quickSort(customerList);  
    }  
}
```

Но что произойдет, если вам нужно изменить эту ссылку в будущем?

Здесь у вас есть ссылка на конкретный класс, называемый `quicksorting`, который вы используете для сортировки списка, который ваша система принимает от пользователя.

Проблема в том, что, если вы должны реализовать другой класс сортировки, называемый `mergesort`, вам нужно внести существенные изменения в ваш класс подсистемы клиента.

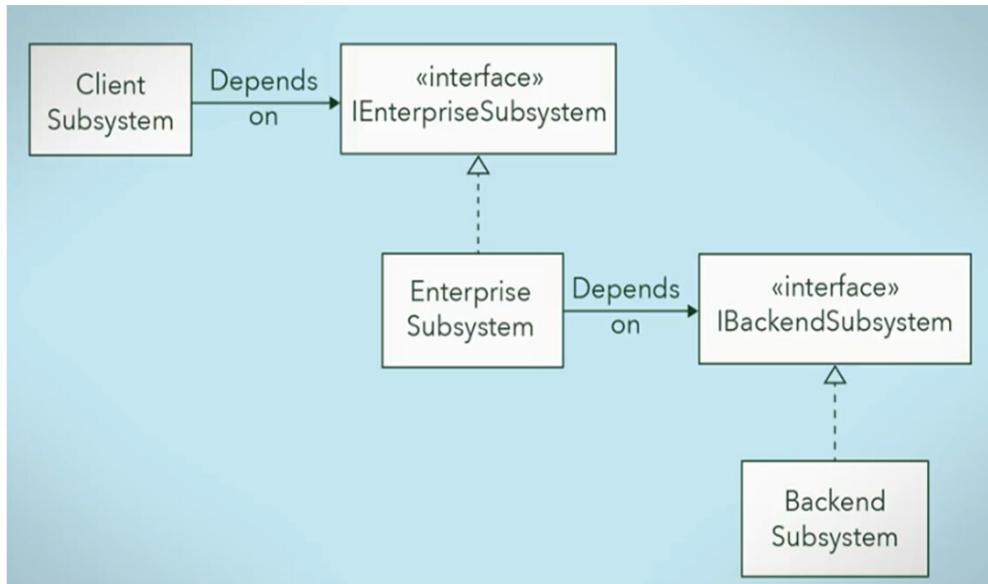
Обратите внимание, что, если вы измените тип класса сортировки от `quicksorting` до `mergesorting`, вам также нужно изменить вызов метода сортировки.

Представьте себе, сколько работы вам нужно будет выполнить каждый раз, когда вы хотите, чтобы ваша система изменила алгоритм сортировки.

Мало того, что это непрактично, такие изменения могут иметь неожиданные побочные эффекты, если вы пропустите какие-либо старые ссылки.

Принцип инверсии зависимостей решает эту проблему, обобщая функциональность низкого уровня в интерфейсы или абстрактные классы, так что, когда предлагаются альтернативные варианты реализации, их можно легко использовать.

Когда вы используете принцип инверсии зависимостей, общая архитектура вашей системы будет очень похожа на модели шаблонов проектирования.



Вам нужно обобщить поведение каждой подсистемы в интерфейс.
Тогда конкретные классы каждой подсистемы будут реализовывать интерфейс.
И, наконец, ваши клиентские классы будут ссылаться на интерфейс, а не напрямую на конкретные классы.

```
public class ClientSubsystem {  
    public Sorting enterpriseSorting;  
  
    public ClientSubsystem(Sorting concreteSortingClass) {  
        this.enterpriseSorting = concreteSortingClass;  
    }  
  
    public void sortInput(List customerList) {  
        this.enterpriseSorting.sort(customerList);  
    }  
}
```

Как вы можете видеть, вместо объявления конкретного класса вы можете объявить интерфейс сортировки.

Затем вы можете определить, какой экземпляр класса сортировки используется клиентской подсистемой во время создания экземпляра.

Кроме того, так как вы можете обобщить поведение сортировки в метод, называемый `sort` в интерфейсе, вам не нужно будет изменять вызов метода в клиентской подсистеме.

Поскольку ваш клиентский класс зависит от обобщения на высоком уровне, а не от конкретного класса низкого уровня, вы можете легко изменить ресурс, который будет использоваться вашим клиентом.

Зависимость высокого уровня не позволит вашему клиентскому классу быть непосредственно связанным с конкретным классом.

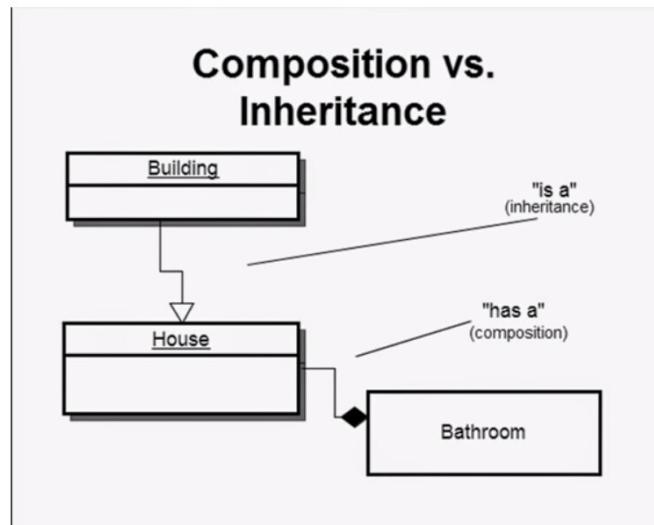
Это означает, что клиентский класс зависит от ожидаемого поведения, а не от конкретной реализации поведения.

Большие системы трудно поддерживать, и если ваша система будет слишком сильно связана, будет трудно вносить в нее какие-либо изменения.

И принцип инверсии зависимостей позволяет вам решить эту проблему и изолировать части вашей системы от конкретных реализаций функциональности.

Принцип композиции объектов

Общей проблемой в объектно-ориентированных системах, к которым пытаются применить шаблоны проектирования, является то, как уменьшить количество жестких связей.



Принципы проектирования, которые используются всеми шаблонами проектирования, определяют разные способы управления связыванием.

Меньшее связывание в вашей системе означает, что ваша система может быть более гибкой и способна обрабатывать изменения существующего базового кода, например, интеграцию новых функций.

Шаблоны проектирования, которые мы рассматривали, широко используют обобщение, абстрагирование и полиморфизм как способы достичь слабого связывания.

В то время как наследование – это отличный способ создать большое количество повторно используемого кода, наследование связано с созданием тесных связей суперклассов с их подклассами.

Почему эта связь является сильной?

Подумайте о том, что такое наследование в объектно-ориентированном программировании.

Когда подкласс наследуется от суперкласса, подкласс получает знания и доступ ко всем атрибутам и методам суперклассов, если их модификаторы доступа не являются приватными.

Это означает, что если у вас есть несколько уровней наследования, подкласс внизу дерева наследования может потенциально иметь доступ к атрибутам и поведениям всех суперклассов.

Что мы можем сделать, чтобы этого избежать?

Вы можете использовать принцип композиции объектов для получения большого количества повторно используемого кода без использования наследования.

Этот принцип гласит, что классы должны использовать повторное использование кода посредством агрегации, а не наследования.

Шаблоны проектирования, такие как шаблон композиции и шаблон декоратора, используют этот принцип проектирования.

Оба этих шаблона составляют конкретные классы для создания более сложных объектов.

Общее поведение создается из суммированного поведения отдельных объектов.

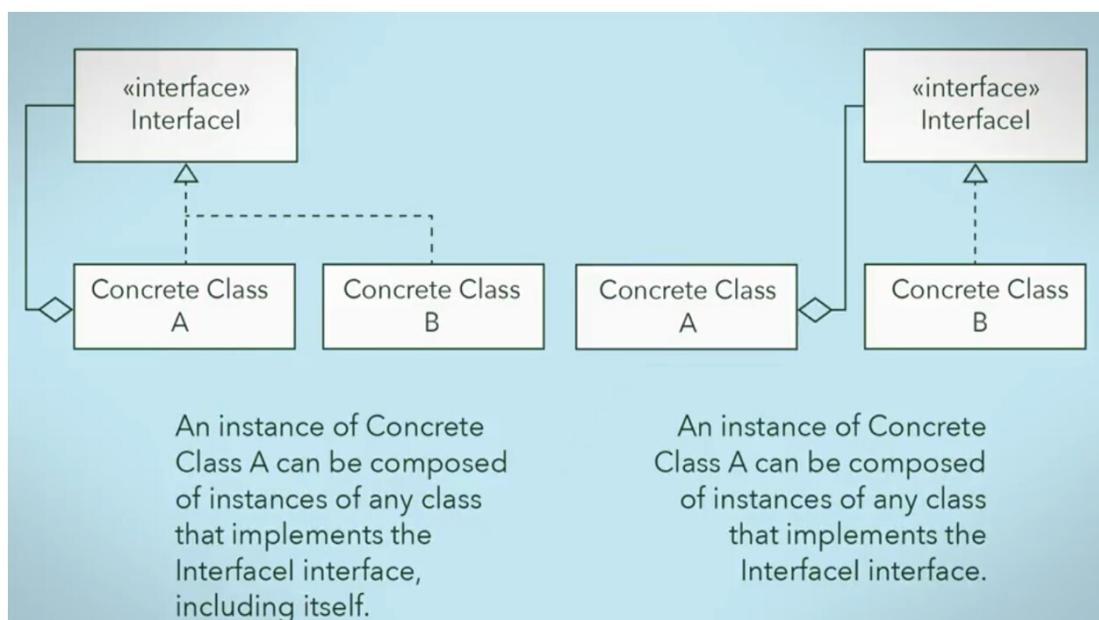
Объект может повторно использовать и агрегировать другой объект, чтобы делегировать ему определенный запрос.

Идея состоит в том, чтобы создать систему, в которой конкретные классы могли бы делегировать задачу другим конкретным классам.

Делегация обеспечит меньший уровень связывания по сравнению с наследованием.

Преимущества очевидны, но и есть недостаток в использовании принципа композиции объектов.

Давайте сначала рассмотрим преимущества.



Мы уже обсуждали основное преимущество принципа композиции объектов, которое заключается в том, что агрегирование и делегирование обеспечивают меньшее связывание, чем наследование.

Поскольку композитные классы не разделяют атрибуты или реализации поведения, они более независимы друг от друга.

Они имеют отношения длины руки, что позволяет легче делать изменения.

В наследовании все подклассы суперкласса тесно связаны с суперклассом.

Как ваши классы будут составлять себя, определяется на этапе проектирования.

На UML диаграмме классов слева объекты могут быть составлены рекурсивно и единообразно.

Или справа, объект состоит из других объектов, имеющих согласованный тип.

Составные объекты обеспечивают большую гибкость вашей системы.

На этапе проектирования легче и естественнее сохранять классы отдельно.

Составные объекты не заставляют вас пытаться найти общие черты между двумя классами и объединять их вместе, как с наследованием.

Вместо этого вы можете создавать классы, которые могут работать вместе, не используя ничего общего между ними.

Эта гибкость помогает при изменениях.

Так как если вы используете наследование, при изменениях, вам, возможно, придется перестроить дерево наследования.

И наконец, компоновка объектов позволяет, по сути, динамически изменять поведение объектов во время выполнения.

Вы можете создать новую общую комбинацию поведения, компонуя объекты.

С наследованием поведение ваших классов определяется во время компилирования.

Это означает, что во время работы программы они не могут изменить поведение.

Существует огромное количество преимуществ, которые композиция имеет по сравнению с наследованием, но вам также нужно знать и о недостатках.

Самый большой недостаток композиции заключается в том, что вы должны обеспечить реализацию для любого поведения без преимущества наследования для совместного использования кода.

Это означает, что у вас будут очень похожие реализации по классам.

При наследовании вам не нужно предоставлять каждому подклассу собственную реализацию общего поведения.

Общая реализация доступна в суперклассе.

Необходимость предоставления реализаций для каждого класса означает, что для этого потребуется время и ресурсы.

Нужно будет предоставить несколько реализаций одного и того же поведения.

Однако при всем при этом, принцип композиции объектов имеет много преимуществ перед использованием наследования для повторного использования кода и гибкого построения поведения.

Этот принцип проектирования помогает уменьшить связывание в системе, используя делегирование и компоновку объектов.

Но при этом нельзя сказать, что наследование никогда не должно использоваться.

Как композиция объектов, так и наследование оба имеют свое место.

Композиция дает лучшую гибкость и меньшее связывание, сохраняя возможность повторного использования, но это не значит, что композиция всегда должна использоваться вместо наследования.

Необходимо изучить требования вашей системы, чтобы определить, какой принцип проектирования лучше подходит.

Это такие вопросы как.

У вас есть набор связанных классов или несвязанных классов?

Какое общее поведение между ними?

Вам нужны специализированные классы для обработки конкретных случаев или вам просто нужна другая реализация того же поведения?

Это вопросы, которые нужно задать при разработке программных систем.

И после этого выбрать соответствующий принцип проектирования.

Разделение интерфейса

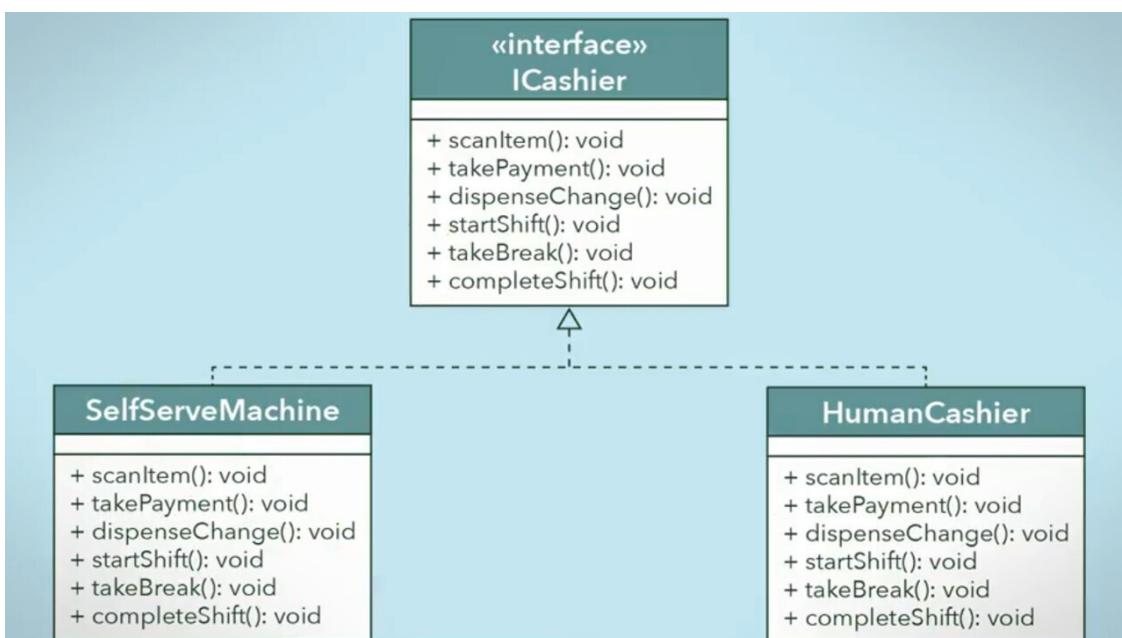
Многие шаблоны проектирования, которые мы рассмотрели, используют обобщения конкретных классов.

Эти обобщения представлены как интерфейсы для клиентских классов, которые используют их для вызова поведения конкретных классов.

Шаблоны проектирования делают это для того, чтобы клиентские классы были менее зависимыми от конкретных классов, тем самым позволяя легче вносить изменения в систему.

Интерфейсы играют важную роль в объектно-ориентированном программировании, поэтому вы должны стремиться программировать интерфейсы вместо конкретных классов.

Однако проблема может возникнуть, если интерфейс содержит слишком много поведений.



Рассмотрим этот пример.

Давайте представим систему оплаты в продуктовом магазине.

В североамериканских странах есть два способа оплаты клиентом товара.

Клиент может подойти к кассиру-человеку, или использовать автоматизированную машину для самообслуживания.

Их работа может быть обобщена с помощью интерфейса.

Они оба просматривают товар, который клиент хочет приобрести и принимают какую-то форму оплаты.

Но также есть некоторые варианты поведения, которые не являются общими для них.

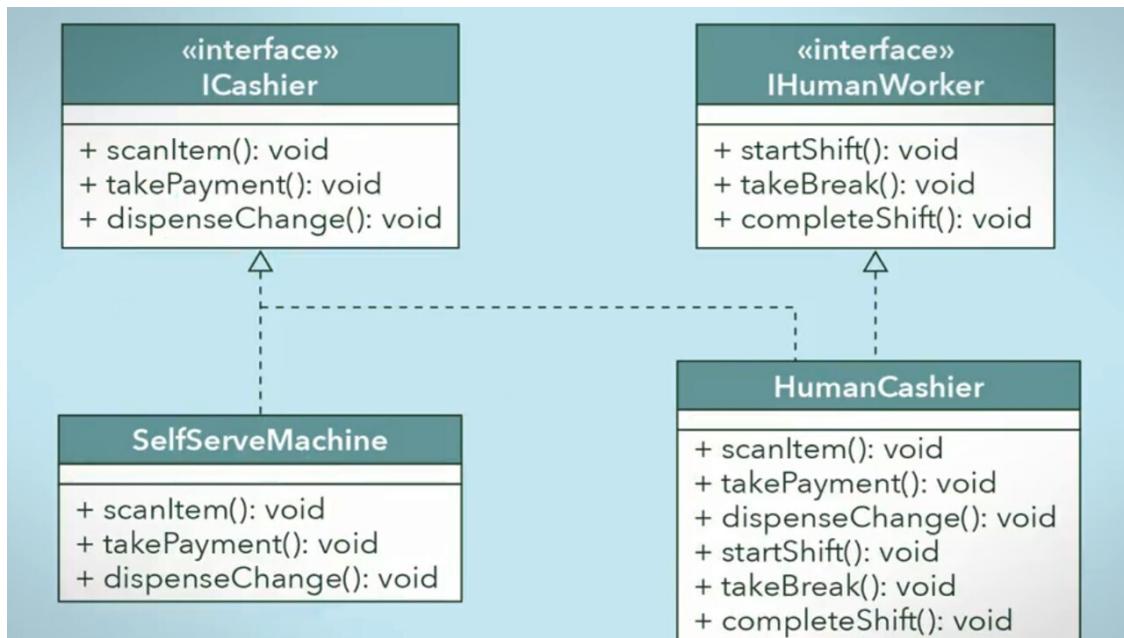
В то время как машина самообслуживания может выполнять работу непрерывно, кассир должен есть, делать перерывы и уходить в конце своей смены.

И вы можете просто поместить эти человеческие поступки в конкретный класс кассира, но при этом если клиентский класс будет вызывать эти методы непосредственно из класса кассира, он станет зависимым от этого конкретного класса.

Поэтому, вы вынуждены будете добавить это человеческое поведение в интерфейс.

Или вы просто согласитесь с тем, что ваша система будет сильно связанной.

К счастью, вы можете использовать принцип разделения интерфейса для решения этой проблемы, чтобы не столкнуться с проблемой зависимостей и не обобщать все в интерфейс.



Принцип разделения говорит, что класс не должен зависеть от методов, которые он не использует.

Это означает, что любые классы, реализующие интерфейс, не должны иметь фиктивных реализаций методов, определенных в интерфейсе.

Вместо этого вы должны разделить большие интерфейсы на более мелкие интерфейсы.

Поэтому мы должны применить принцип разделения интерфейса и разделить интерфейс ICashier на два меньших интерфейса.

Это позволит каждому интерфейсу более точно описать ожидаемое поведение, и это позволит выбрать правильные комбинации интерфейсов, которые должен реализовать конкретный класс.

Таким образом, в этом примере оба конкретных класса будут обеспечивать реализацию только тех интерфейсов, которые обобщают их конкретные функции.

Интерфейсы являются неотъемлемой частью объектно-ориентированных систем.

Они помогают уменьшить связывание, обобщая конкретные классы, но, как и со всеми объектно-ориентированными принципами, вы должны стремиться их правильно использовать.

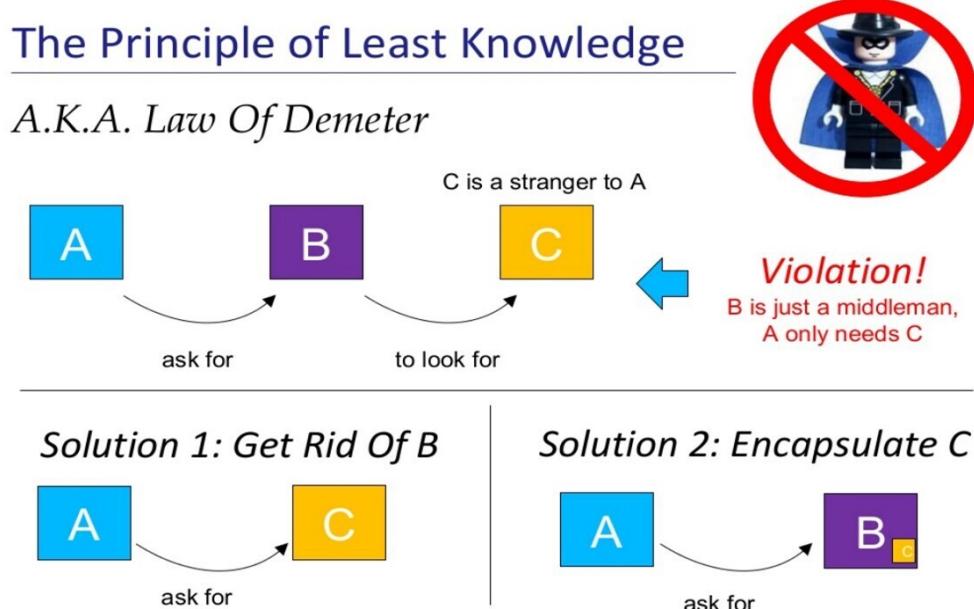
В то время как большие интерфейсы не являются плохим решением дизайна, тем не менее вам необходимо изучить их в контексте вашей системы, чтобы решить, нужно ли разбить их на более мелкие обобщения с использованием принципа разделения.

Этот принцип проектирования говорит, что класс не должен зависеть от методов, которые он не использует, а интерфейсы должны быть разделены таким образом, чтобы они могли правильно описать отдельные функции вашей системы.

Принцип наименьшего знания

Чтобы справиться с сложностью системы, одна из идей заключается в том, что классы системы должны быть спроектированы таким образом, чтобы они не нуждались в знаниях и максимально не зависели от других классов в системе.

Это уменьшает связывание и упрощает поддерживаемость системы.



Ограничевая общение классов между собой, вы можете обеспечить соблюдение принципа наименьшего знания в своей системе.

Этот принцип также реализуется в правиле, известном как Закон Деметры.

Основная идея этого правила заключается в том, что классы должны знать и взаимодействовать с как можно меньшим количеством других классов.

Это означает, что любой класс должен общаться только со своими ближайшими соседями.

Этими соседями должны быть другие классы, о которых должен знать только один класс.

Этот принцип является немного более абстрактным, чем другие принципы проектирования, но он также важен, так как он помогает уменьшить связывание и обеспечить стабильность системы.

Закон Деметры фактически состоит из разных правил.

Правила в Законе Деметры используются для определения того, как один метод может вызывать другой метод.

```
public class Friend {  
    public void N() {  
        System.out.println("Method N invoked");  
    }  
}  
  
public class O {  
    public void M(Friend P) {  
        P.N();  
        System.out.println("Method M invoked");  
    }  
}
```

Первое правило гласит, что метод **M** в объекте **O** может вызвать любой другой метод внутри самого объекта **O**.

Это правило довольно простое и имеет логический смысл.

Метод, инкапсулированный внутри класса, разрешен для вызова любого другого метода, который также инкапсулирован в пределах этого класса.

В этом примере методу **M** разрешено вызывать метод **N**, и оба метода являются частью одного и того же класса.

```
public class Friend {  
    public void N() {  
        System.out.println("Method N invoked");  
    }  
}  
  
public class O {  
    public void M(Friend P) {  
        P.N();  
        System.out.println("Method M invoked");  
    }  
}
```

Второе правило гласит, что метод **M** может вызывать методы любого параметра **P**.

Поскольку параметр метода считается локальным для метода, параметр можно считать ближайшим соседом.

Поэтому можно вызвать методы параметра в классе.

Метод **M** в классе **O** имеет параметр **P** типа **Friend**.

Закон Деметры позволяет в методе M вызывать любой метод класса Friend.

```
public class Friend {  
    public void N() {  
        System.out.println("Method N invoked");  
    }  
}  
  
public class O {  
    public void M() {  
        Friend I = new Friend();  
        I.N();  
        System.out.println("Method M invoked");  
    }  
}
```

Третье правило Закона Деметры заключается в том, что метод M может вызывать метод N объекта I, если объект I создан в M.

Это означает, что если метод создает новый объект, то этот метод может использовать методы нового объекта.

Так как объект считается локальным для этого метода.

Точно так же как объект считается локальным, когда он является параметром.

```
public class Friend {  
    public void N() {  
        System.out.println("Method N invoked");  
    }  
}  
  
public class O {  
    public Friend I = new Friend();  
    public void M() {  
        this.I.N();  
        System.out.println("Method M invoked");  
    }  
}
```

В дополнение к локальным объектам в четвертом правиле Закона Деметры указывается, что любой метод M в объекте O может вызывать методы любого объекта, который является прямым компонентом O.

Это означает, что метод класса может вызывать методы его переменных экземпляра.

Так как класс O имеет прямую ссылку на класс Friend, любому методу внутри O разрешено вызывать любой метод класса Friend.

Закон Деметры кажется сложным и абстрактным принципом, но все правила сводятся к правилу, согласно которому вы не должны позволять методу доступ к другому методу путем доступа к другому объекту.

Это означает, что метод не должен вызывать методы любого объекта, который не является локальным.

Мы только что посмотрели, что Закон Деметры считает локальным объектом.

Эти объекты должны передаваться через параметр или они должны быть созданы в рамках метода или должны быть переменными экземпляра.

Это позволяет методам иметь прямой доступ к поведению локальных объектов.

Нарушения Закона Деметры обычно возникают, когда у вас есть цепочка вызовов методов объектов, о которых вы не должны знать или, когда используются методы неизвестного типа объекта, который возвращается из вызова метода.

```
public class Driver {  
    Car myCar = new Car();  
  
    public void drive() {  
        this.myCar.engine.start();  
    }  
}
```

Это пример вызова метода объекта, который не принадлежит данному классу.

Водитель знает о классе автомобиля, но двигатель автомобиля не является компонентом водителя.

Поэтому водитель не должен вызывать методы двигателя.

Вам не следует обращаться к методам объектов, которые не находятся в ваших ближайших соседях.

Вы просто должны сказать машине ехать, и сам автомобиль должен знать, как обращаться с его компонентами.

```
public class Driver {  
    public Car myCar;  
  
    public void rentVehicle(VehicleRentalStore store) {  
        Motorcycle myRental = store.rent("motorcycle");  
        myRental.drive();  
    }  
}
```

Другой способ, которым вы можете нарушить Закон Деметры, – это когда ваш метод получает объект неизвестного типа в качестве возвращаемого значения, и вы вызываете методы возвращаемого объекта.

Возвращаемые объекты должны быть того же типа, что и те, которые указаны в параметре метода, или те, которые объявлены и созданы локально в методе, или те, которые объявлены в переменных экземпляра класса, который инкапсулирует метод.

Закон Деметры определяет, как классы должны взаимодействовать друг с другом.

Они не должны иметь полный доступ ко всей системе, поскольку это создает высокую степень связывания.

То есть классы должны знать как можно меньше о системе в целом.

Анти-паттерны

Иногда, независимо от того, насколько хорошо вы разрабатываете свой код, все еще будут изменения, которые необходимо внести.

Трудно, если не невозможно, сделать это правильно за один раз.

Именно здесь и происходит рефакторинг.

Рефакторинг – это процесс внесения изменений в код, так что внешнее поведение кода не изменяется, но внутренняя структура кода улучшается.

Это делается путем внесения небольших, необязательных изменений в структуру кода и частого тестирования, чтобы убедиться, что эти изменения не изменили поведение кода.

В идеале, вам не нужно проводить рефакторинг после того, как ваш код завершен.

Это может занять много времени и может вызвать больше проблем, чем вы хотите исправить.

Обычно рефакторинг проводится при добавлении функциональности.

Рефакторинг кода на этом этапе может сделать добавление более легким.

Какие изменения вносятся путем рефакторинга?

Подобно тому, как возникают шаблоны проектирования, появляются шаблоны неправильного кода.

Они называются анти-шаблонами или анти-паттернами проектирования.

Наиболее распространенный пример анти-паттерна касается комментариев.

Отсутствие комментариев

Слишком много комментариев

Обычно проблема заключается в том, что в коде нет комментариев.

Отсутствие комментариев затрудняет понимание того, что делает код.

Без комментариев плохо, но и слишком много комментариев тоже может вызвать проблемы.

При изменении кода они могут запутать понимание кода.

Многие разработчики пытаются скрыть плохой дизайн системы, используя много комментариев, которые объясняют, что делает код.

Скорее всего, если вы видите много комментариев, которые описывают сложный дизайн, они скрывают плохой дизайн.

Также использование комментариев для объяснения дизайна иногда может указывать на то, что язык программирования не подходит для кодирования данной системы.

Возможно, язык программирования не поддерживает принципы дизайна, которые вы пытаетесь применить.

Например, в первые годы существования Java, еще не существовала концепция дженериков.

Разработчики должны были комментировать, чтобы объяснить, что они делают с кодом при приведении типов.

В итоге дженерики были встроены в Java.

Но до этого приходилось использовать много комментариев.

С другой стороны, комментарии очень полезны для документирования программного интерфейса.

А также для документирования обоснования того, почему была выбрана конкретная структура данных или алгоритм.

И комментарии также позволяют другим разработчикам легко использовать ваш код.

Как и для большинства вещей в программировании, существует баланс, который нужно найти для эффективного использования комментариев в коде.

Дублирование кода

Следующий анти-паттерн, – это дублированный код.

Дублированный код – это когда у вас есть блоки кода, которые похожи, но имеют небольшие отличия.

Эти блоки кода присутствуют в нескольких местах вашего программного обеспечения.

Теперь, по мере расширения системы, вы также хотите добавить некую функциональность.

Чтобы добавить эту функциональность, вам нужно обновить код во всех местах.

И при этом вы можете пропустить какое-либо место и тем самым нарушить систему.

Поэтому необходимо объединить весь этот код в одном месте.

Длинный метод

Большой класс

Слишком много небольших классов

Еще одна проблема – это использование длинных методов.

Это относительно очевидно.

Вам не нужно использовать длинные методы.

Наличие длинного метода иногда может указывать на то, что в этом методе больше, чем должно быть.

Или это сложнее, чем это нужно.

Но какая длина должна быть у метода?

Нет волшебного правила, в котором говорится, что метод с определенным количеством строк является слишком длинным.

Например, методы, которые настраивают пользовательский интерфейс, могут быть длинными, несмотря на то, что они сосредоточены на одной конкретной задаче.

Некоторые разработчики считают, что наличие метода, видимого сразу на экране, является хорошим ориентиром, и такой метод состоит не более чем из 50 строк кода.

Подобно наличию больших методов, наличие больших классов также является проблемой.

Эти большие классы обычно называются Blob классами или классами Черной дыры.

Обычно они начинаются с обычного размера.

Но со временем требуется больше функций, и эти классы кажутся подходящим местом для выполнения этих функций.

И эти новые функции, как правило, привлекают еще больше функций.

И класс начинает разрастаться.

И как черная дыра, класс становится все больше и больше и продолжает привлекать все больше и больше функциональности.

В конце концов вам понадобятся обширные комментарии для описания, где в коде класса есть определенные функциональные возможности.

И снова комментарии могут быть индикатором плохого дизайна.

Чтобы этого избежать, вам нужно четко указать цель класса и сохранять логику класса.

Если функциональность прямо не относится к ответственности класса, вам нужно разместить ее где-то еще.

И напротив, наличие слишком большого количества небольших классов также является проблемой, и часто это классы данных.

Классы данных – это классы, которые содержат только данные и не имеют реальной функциональности.

Как правило, эти классы имеют методы getter и setter, но не более того.

Примером класса данных может служить класс двумерной точки, который просто содержит координаты x и y.

Другие классы могут манипулировать этими данными, создавая поведения.

И возможно некоторые из поведений лучше разместить в одном из классов данных.

Например, в классе двумерной точки могут быть различные функции преобразования, которые перемещают точку.

Другая похожая проблема, – это компоненты данных.

Компоненты данных – это группы данных, которые отображаются вместе в переменных экземпляра класса или параметрах для методов.

```
public void doSomething (int x, int y, int z) {
```

```
    ...
```

```
}
```

```
public void doSomething (Point3D point) {
```

```
    ...
```

```
}
```

Скажем, у нас есть метод, который делает что-то с целыми переменными x, у и z.

Теперь, если у нас есть много методов, которые выполняют различные манипуляции с этими переменными.

Было бы разумнее иметь объект как параметр, а не использовать эти переменные в качестве параметров снова и снова.

В этом примере мы могли бы использовать 3D-объект в качестве параметра, который содержит значения x, у и z.

Теперь, вместо использования трех значений данных в качестве параметров, они теперь хранятся внутри объекта.

Наличие длинного списка параметров может затруднить использование метода.

Имея длинный список параметров, вы увеличиваете вероятность того, что что-то пойдет не так.

Как правило, такие методы требуют обширных комментариев, чтобы объяснить, что делает каждый из параметров.

И, как мы обсуждали ранее, наличие большого количества комментариев, как правило, признак плохого дизайна.

Поэтому лучшим решением для длинных списков параметров является введение объектов параметров.

Теперь давайте немного поговорим о некоторых проблемах, которые возникают, когда вы вносите изменения в код.

Расходящиеся изменения

Расползающееся изменение

Первая проблема называется расходящимся изменением.

Эта проблема возникает, когда вам приходится изменять класс разными способами по разным причинам.

Эта проблема тесно связана с проблемой большого класса.

Когда у вас есть большой класс, у него будет много разных функций.

И может потребоваться изменить эти функции различными способами для различных целей.

Таким образом, плохое разделение задач является общей причиной расходящихся изменений.

Хорошо, когда у вашего класса есть только одна конкретная цель.

Это уменьшит количество причин, по которым код необходимо будет изменить.

И в результате уменьшится разнообразие изменений, которые вам нужно реализовать.

Если вы обнаружите, что вы изменяете класс несколькими способами, это может быть хорошим показателем того, что функциональность класса должна быть разбита на отдельные классы.

И первоначально большой класс затем делегирует ответственность этим извлеченным классам.

Таким образом, вы решите две проблемы – большого класса и расходящихся изменений.

С другой стороны, предположим, что вы хотите сделать одно изменение.

Вы хотите реализовать одно небольшое требование, и было бы хорошо, если бы это изменение было в одном месте.

Но вдруг оказывается, что это изменение затрагивает целую группу классов.

Это часто встречающаяся проблема.

Изменение в одном месте требует от вас исправить множество других областей кода.

Это может произойти, когда вы пытаетесь добавить функцию, исправить ошибки или изменить алгоритм.

В идеале вы хотите, чтобы ваши изменения были локализованы. Но это не всегда возможно.

Если вам нужно внести изменения по всему вашему коду, скорее всего, вы что-то пропустите или создадите проблемы в другом месте.

Обычно эту проблему можно решить, перемещая методы.

Если вы обнаруживаете, что изменение требует от вас внесения изменений во множество методов во многих разных классах, то это может быть индикатором того, что эти методы было бы лучше объединить в один или два класса.

Тот факт, что изменение в одном месте приводит к изменениям в других местах, показывает, что эти методы каким-то образом связаны.

И, возможно, есть лучший способ организовать их.

Функциональная зависимость

Неприемлемая близость

Следующий анти-паттерн – это функциональная зависимость.

Функциональная зависимость возникает, когда у вас есть метод, который больше интересуется деталями другого класса, чем тем классом, в котором он находится.

Если кажется, что два метода или классы всегда взаимодействуют друг с другом и должны быть вместе, тогда скорее всего, они должны вместе.

Если у вас есть метод в классе, который, похоже, хочет много общаться с другим классом, чтобы выполнить свою работу, может быть лучше, чтобы он был в пределах этого другого класса.

Другой похожий анти-паттерн – это неприемлемая близость.

Это когда два класса слишком сильно зависят друг от друга посредством двусторонней коммуникации.

Скажем, у вас есть два класса, которые тесно связаны друг с другом.

То есть, метод в одном классе вызывает методы другого класса и наоборот.

Должны ли эти классы быть настолько тесно связаны?

Возможно нет.

Чтобы удалить это связывание, вы можете выделить методы, которые эти оба класса используют, в другой класс.

Двухсторонняя коммуникация не обязательно является плохой.

Есть ситуации, в которых такие коммуникации необходимы.

Однако, если есть способ удалить двухстороннюю коммуникацию, это сделает ваш проект более простым и понятным.

Ранее мы говорили о Законе Деметры или о принципе наименьшего знания.
Этот принцип определяет, какие методы вы можете вызывать.

Цепочки сообщений

```
a.getB().getC().doSomething();
```

И следующий анти-паттерн, – это цепочки сообщений, который нарушает закон Деметры.
Скажем, у вас есть объект A, у которого есть метод getB, и он возвращает объект B.
В этом объекте B вы вызываете метод getC, который возвращает объект C.
И наконец, в этом объекте C вы вызываете нечто, чтобы что-то сделать.
Длинные цепочки сообщений создают сильную связанность и сложность дизайна.
Это также делает код более сложным для тестирования.
Вам нужно перемещаться по цепочке и зависимостям этих объектов.
Скажем, вы хотите переделать свой дизайн и перестроить эти зависимости.
Если вы это сделаете, вы сломаете этот код.
С другой стороны, если вы видите строку кода с цепочкой вызовов, это не всегда плохо.
Если все эти объекты в цепочке следуют Закону Деметры, тогда это может быть хорошо.

Примитивная одержимость

Следующий анти-паттерн – это примитивная одержимость.

Это когда вы слишком много полагаетесь на использование встроенных типов.

Эти встроенные типы или примитивные типы int, long, float или string.

Очевидно, что их нужно использовать.

Тем не менее, они должны присутствовать на самых низких уровнях вашего кода.

Чрезмерное использование примитивных типов происходит, когда вы не определяете подходящие классы.

Например, вы можете просто определить или закодировать все с помощью строк и поместить их в массивы.

И ваш код станет похожим на тот, который разрабатывался в 60-х годах.

Однако с тех пор языки программирования сильно продвинулись, чтобы позволить нам определять наши собственные типы.

И вы должны это использовать.

В противном случае у вас будет не объектно-ориентированный способ мышления.

Например, в коде вы используете почтовые индексы.

Вы можете использовать в коде все эти наборы символов, или вы можете определить класс почтовых индексов и использовать его, вместо массивов строк.

Также помимо простого хранения, вы можете включить в этот класс, например, методы проверки почтового индекса, и другое поведение.

switch

```
public class Animal {  
    private final static int DOG = 0;  
    private final static int CAT = 1;  
  
    private int type;  
  
    ...  
  
    public void say() {  
        switch (type) {  
            case DOG:  
                System.out.println("Woof!");  
                break;  
            case CAT:  
                System.out.println("Meow!");  
                break;  
        }  
    }  
}
```

Следующий анти-паттерн – это switch выражение.
Вы можете подумать, что случилось с оператором switch?
Почему он существует, если он плохо разработан?
Иногда в вашем коде нужно иметь большие, длинные, условные выражения.
И бывают случаи, когда операторы switch могут обрабатываться лучше другим способом.
Например, если ваши условные выражения проверяют типы чего-либо, тогда существует лучший способ обработки оператора switch.
Здесь, в этом примере, метод say проверяет тип животного.
И вы хотите сократить эти условные условия до дизайна, который использует полиморфизм.

```
public interface Animal {  
    public void say();  
}  
  
public class Dog implements Animal {  
    ...  
    public void say() {  
        System.out.println("Woof!");  
    }  
}  
  
public class Cat implements Animal {  
    ...  
    public void say() {  
        System.out.println("Meow!");  
    }  
}
```

В этом улучшенном дизайне у нас есть объекты собаки и кошки, у которых есть своя реализация метода say.

Спекулятивное обобщение

Отказ от запроса

Следующий анти-паттерн – это спекулятивное обобщение.

Спекулятивное обобщение возникает, когда вы делаете суперкласс или интерфейс, который не нужен сейчас, но вы думаете, что можете использовать его когда-нибудь.

Если вы это делаете, вы вводите обобщение, которое не может реально помочь коду.

С гибкой разработкой вы должны практиковать дизайн, тот который нужен именно сейчас.

У вас должен быть дизайн, достаточный, чтобы удовлетворить требованиям конкретной итерации рабочего цикла разработки.

В начале каждой итерации разработки вы должны выбрать набор требований, которые будут разрабатываться.

Это все, что вам нужно для разработки.

И вы можете игнорировать все остальные требования.

Как вы знаете, программное обеспечение часто меняется.

И ваш клиент может изменить свое мнение в любое время и отказаться от каких-либо требований.

Поэтому ваш дизайн должен оставаться простым.

Вам не нужно тратить время на написание кода, который может никогда не понадобиться.

И наконец, последний анти-паттерн, который мы рассмотрим, – это отказ от запроса.

Это происходит, когда подкласс наследует что-то и не нуждается в этом.

Например, вам завещали набор фарфоровых фигурок.

Тот, кто завещал их вам, собирал их всю жизнь и теперь у вас есть сотни этих фигурок, но они вам не нужны и вам негде их разместить.

То, что предназначалось как подарок, превратилось в большую нагрузку.

Это также происходит и в коде.

Скажем, суперкласс объявил общее поведение во всех его подклассах.

Если вы обнаружите, что эти подклассы наследуют то, что они не используют или в чем они не нуждаются, то целесообразно ли делать эти классы подклассами этого суперкласса?

Возможно, будет иметь смысл сделать их самостоятельными классами или, возможно, это ненужное поведение не должно определяться в суперклассе.

Если только некоторые подклассы могут использовать это поведение, возможно, было бы лучше просто определить это поведение только в этих подклассах.

Вопросы

Вопрос 1

Выберите два элемента принципа открыто-закрытый:

Закрыт для расширения.

Открыт для расширения +

Закрыт на техническое обслуживание.

Открыт для изменения

Открыт для обслуживания

Закрыт для изменения +

Вопрос 2

Какое наилучшее описание принципа инверсии зависимостей?

Объекты клиента зависят от обобщений вместо конкретных объектов. +

Объекты службы подписываются на свои потенциальные клиентские объекты как Наблюдатели, наблюдая за запросом.

Объекты клиента зависят от интерфейса службы, который направляет их запросы.

Объекты клиента зависят от шаблона адаптера для взаимодействия с остальной частью системы.

Вопрос 3

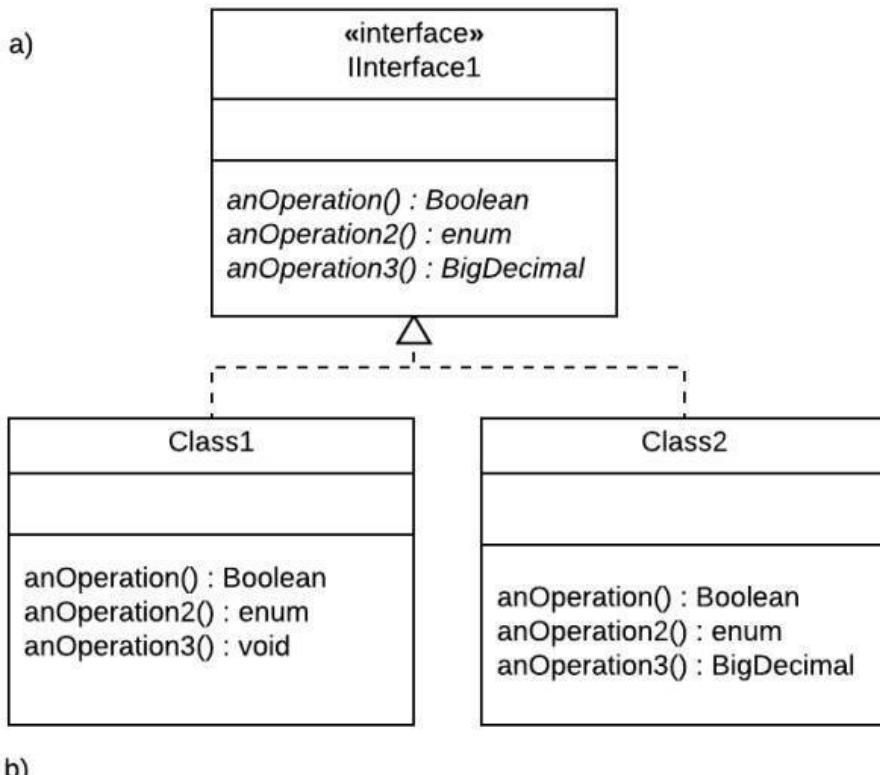
Какое из этих утверждений верно в отношении принципа композиции объектов?

Он обеспечивает поведение с помощью агрегации вместо наследования +

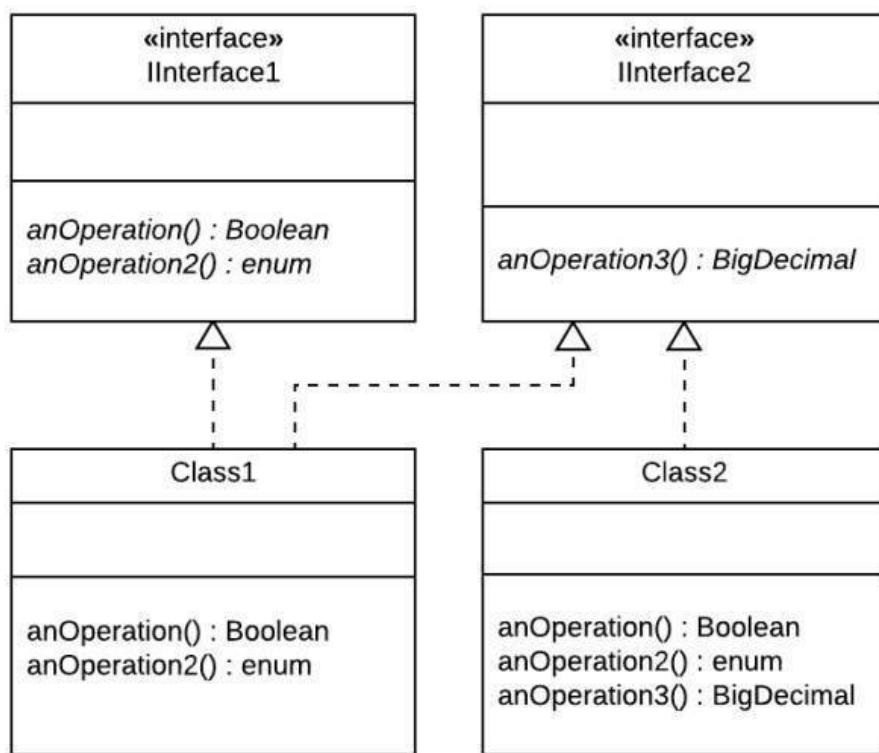
Он приводит к более сильной связи

Вопрос 4

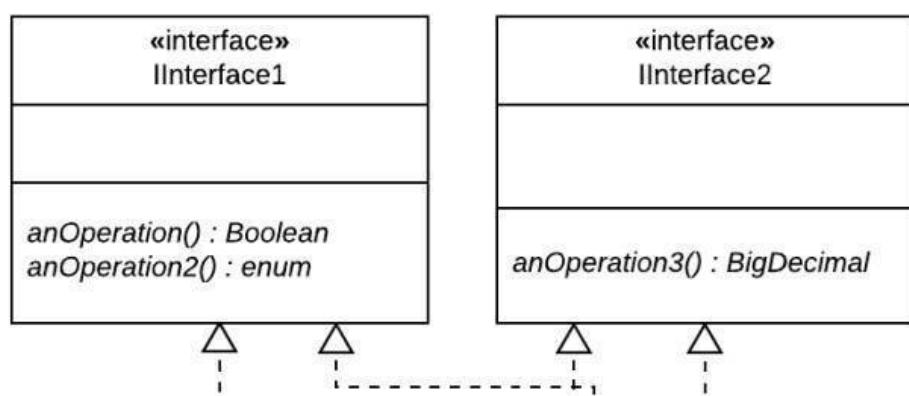
Какая из этих UML-диаграмм демонстрирует принцип разделения интерфейса?



b)



c)



a)

b)

c)

d) +

Вопрос 5

Какой из этих примеров кода нарушает Принцип наименьшего знания или Закон Деметры?

a) **public class O {**

M I = new M();

public void anOperation2() {

this.I.N.anOperation();

}

} +

b) **public class Class1 {**

public void N() {

System.out.println("Method N invoked");

}

}

public class Class2 {

public void M(Class1 P) {

P.N();

System.out.println("Method M invoked");

}

c) **public class O {**

public void M() {

this.N();

System.out.println("Method M invoked");

}

public void N() {

System.out.println("Method N invoked");

}

}

d) **public class P {**

public void N() {

System.out.println("Method N invoked");

}

}

public class O {

public void M() {

P I = new P();

I.N();

System.out.println("Method M invoked");

}

}

Вопрос 6

Когда можно рассматривать комментарии как признак анти-паттерна?

Когда комментарий используется для объяснения обоснования проектного решения

Комментарии помогают уточнить код.

Слишком много комментариев делают файлы слишком большими для компиляции.

Чрезмерное комментирование может скрывать плохой код +

Вопрос 7

Что такое примитивная одержимость?

Код, который содержит много объектов низкого уровня, без использования принципов ООП, таких как агрегация или наследование. +

Чрезмерное использование примитивных типов данных, таких как int, long, float.

Использование множества разных примитивных типов вместо того, чтобы опираться на несколько, которые вместе охватывают соответствующий уровень детализации для вашей системы.

Использование пар ключ-значение вместо абстрактных типов данных.

Вопрос 8

У вас есть класс, в который вы продолжаете добавлять функции. Всякий раз, когда вы добавляете новые функции, это кажется естественным, но это начинает становиться проблемой! Какой это анти-паттерн?

Длинный метод

Большой класс +

Расходящееся изменение

Спекулятивное обобщение

Вопрос 9

Почему важно избегать цепочек сообщений, когда это возможно?

Полученный код обычно является сильно связанным и сложным. +

Если возвращается неожиданный объект, это может привести к ошибкам во время выполнения.

Обходным путем является получение частных методов, которые важны для инкапсуляции.

Это снижает когезию в вашем классе.

Вопрос 10

Посмотрите на фрагмент кода. Какой анти-паттерн вы обнаружите?

```
public class Class1 {
```

```
...
```

```
public void M(Class2 C) {
```

```
C.doSomething(x);
```

```
C.foo(y);
```

```
C.foo2(z, i);
```

```
}
```

```
}
```

Функциональная зависимость +

Длинный список параметров

Расходящееся изменение

Неподходящая близость

Вопрос 11

Джозеф разрабатывал класс для своей игры на смартфоне в покер и решил, что в один прекрасный день он захочет изменить картинку на оборотной стороне карт, поэтому он создал суперкласс класса Deck. Поскольку его приложение еще не имеет этой функции, у Deck есть только один подкласс, RegularDeck. Какой это анти-паттерн?

Расходящееся изменение
Спекулятивное обобщение +
Примитивная одержимость
Отказ от запроса