GeekBrains

Структурное проектирование

Технологическая специализация системный аналитик

Оглавление

Введение

На предыдущих лекциях мы познакомились с концептуальным проектированием, изучили основные этапы построения концептуальной модели, рассмотрели построение функциональной модели деятельности, позволяющей наглядно выделить функции системы, проанализировать их взаимосвязи и назначение.

Цель настоящей лекции – предоставить системному аналитику знания и навыки обращения с инструментарием структурного проектирования, позволяющими построить модель данных.

Термины, используемые в лекции

Система – совокупность взаимодействующих и взаимосвязанных элементов.

Модель системы – формализованное описание системы.

Системный анализ – совокупность приемов научного познания представляющий собой последовательность действий по установлению структурных связей между переменными или элементами исследуемой системы.

Структура — определенная взаимосвязь, взаиморасположение составных частей, строение, устройство чего-либо

Аспект (Точка зрения) – позиция, относительно которой производится описание системы, процесса или явления.

User Story – пользовательская история, служащая для выявления требований пользователя системы

Use Case – варианты использования, служат для выявления функций системы

MVP – минимально жизнеспособный продукт, являющийся концептом системы

Объект – это предмет или явление (процесс), имеющие четко выраженные границы, индивидуальность, поведение и жизненный цикл

Состояние объекта — совокупность значений его свойств (атрибутов) и связями с другими объектами, оно может меняться со временем**.**

Поведение объекта — характерные (присущие) действия объекта и его реакция на запросы от других объектов

Индивидуальность объекта — это свойство, отличающее его от всех других объектов

Прототип (объекта) — это образец, по образу и подобию которого создаются другие объекты

Класс – это шаблон, определяющий общность объектов, которые являются экземплярами класса, в прикладном смысле представляющая группу данных и методов(функций) для работы с этими данными.

Конструктор класса — специальный блок инструкций, вызываемый при создании объекта

Деструктор — специальный метод класса, служащий для деинициализации объекта

Атрибут — поименованное свойство класса, определяющее диапазон допустимых значений, которые могут принимать экземпляры данного свойства

Операция — это услуга, которую можно запросить у любого объекта данного класса

Метод — это реализация операции, определяющая функцию или процедуру принадлежащую конкретному классу или объекту

Дескриптор — это атрибут объекта со связанным типовым поведением с определенными методами

Делегирование – переопределение реализации во время создания экземпляра

Наследование – механизм гарантирования переноса свойств и поведения от предка к потомку

Полиморфизм — возможность использовать одни и те же методы для объектов разных классов

Абстракция — это прием фокусирования на свойствах системы, важных для текущей задачи

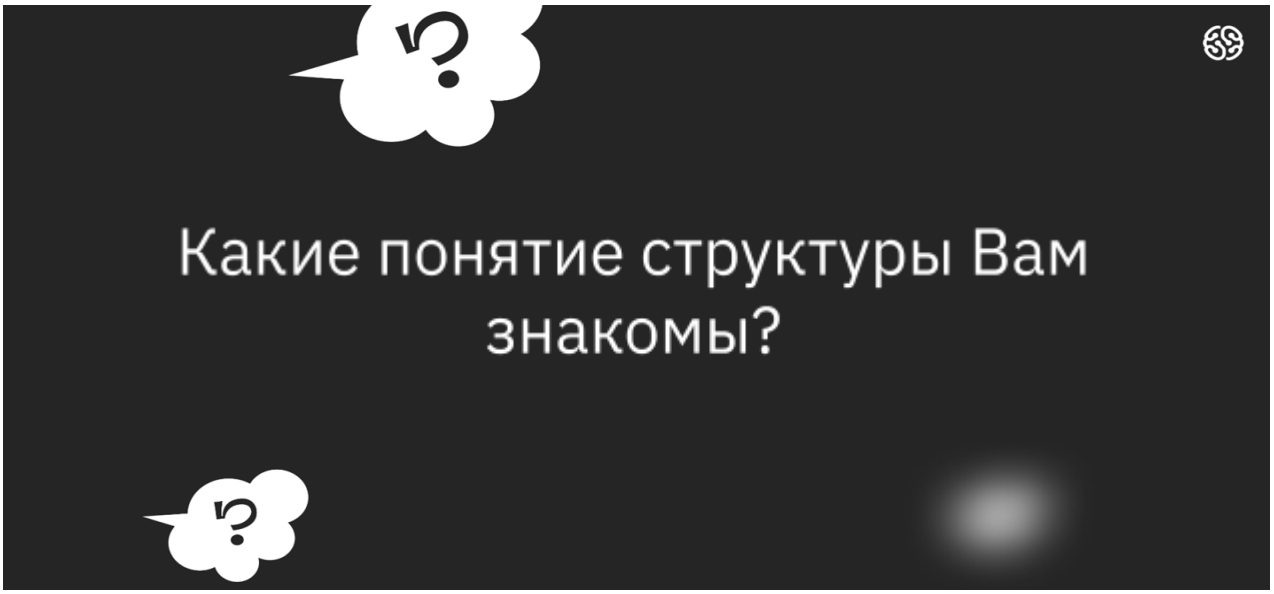
Инкапсуляция — это заключение данных и функциональности в оболочку, защищающую от внешнего воздействия

Интерфейс — это совокупность операций, определяющих набор услуг класса или компонента. Интерфейс не определяет внутреннюю структуру, все его операции открыты

Компонент — это относительно независимая и замещаемая часть системы, выполняющая четко определенную функцию в контексте заданной архитектуры

Роль и место структурного проектирования

Для начала давайте ответим на вопрос о структуре. Постарайтесь вспомнить, что такое структура и какой она бывает в различных областях науки и практики?



Ответ:

В философии, Структура — совокупность связей между частями объекта.
В физике, Структура — группа уровней энергии и спектральных линий, различающихся из-за квантовых взаимодействий.
В химии, структура - представляет собой пространственное упорядочение атомов и связей в молекуле.
В математике, структура — какой-либо новый объект, вводимый на некотором множестве, свойство элементов множества.
В менеджменте, структура - это пакет официальных документов, отражающих иерархию и состав организации, а также функции, права и обязанности ее основных элементов.
В информатике, структура — формат организации, управления и хранения данных, который обеспечивает эффективный доступ и модификацию.
Как мы видим, в каждой предметной области структура имеет собственное определение, общим же остается то, что структура это характеристика взаимосвязи элементов, которая описывает их взаимное расположение.

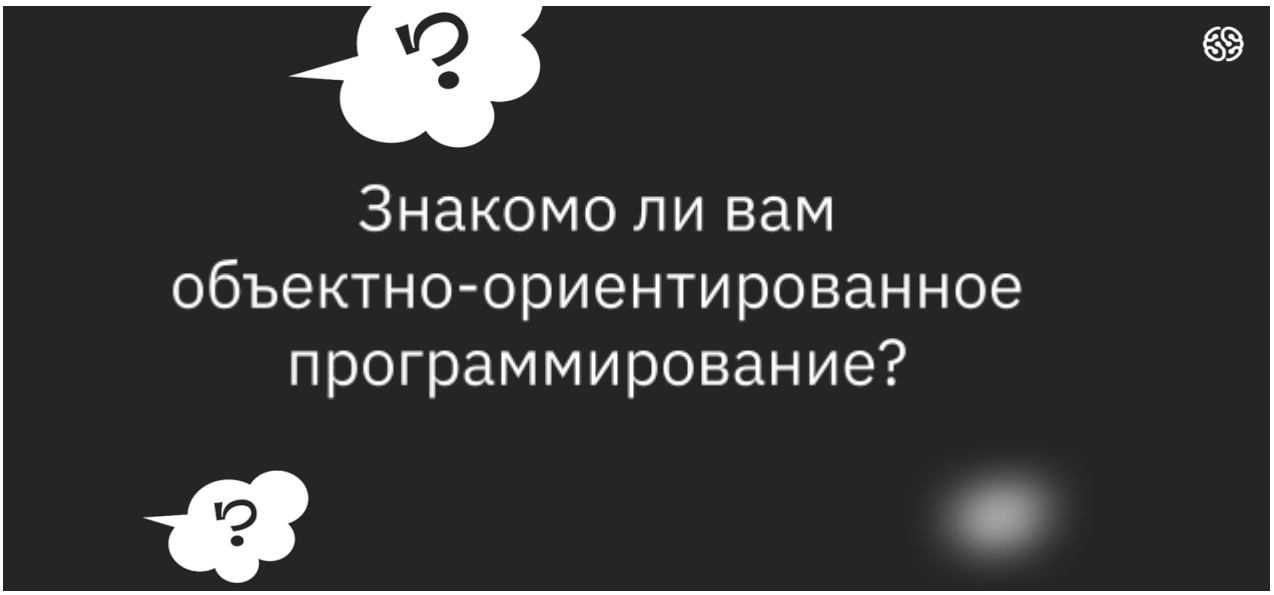
В этом смысле понятие структуры присуще системному анализу в части раскрытия взаимного отношения элементов системы.
В системном анализе понятие структуры используется и в концептуальном и в функциональном описании системы.

Так на предыдущих лекциях мы говорили о модели системы, модели предметной области и функциональной декомпозиции системы. При этом мы неявно затрагивали понятие структуры, так как оно присуще любому описанию взаимного расположения элементов системы на любом уровне абстракции и для любого её аспекта.

Таким образом, тема данной лекции в определенной мере дополнят ранее изложенный материал, но вместе с тем, при проектировании информационных систем, её стоит выделить отдельно ввиду сложности описания, которая явно определяет стиль и особенности будущей реализации системы.

Результатом структурного анализа информационной системы выступает её **объектная модель**, не только определяющая дальнейшую фактическую реализацию системы, но и требующая отдельного описательного инструментария, позволяющего не просто составить проект определяющий требования к реализации системы, но и провести анализ, позволяющий её оптимизировать на низком уровне абстракции, рассматривая отдельные объекты и их свойства.

Эта оптимизация производится с целью упрощения описания системы при сохранении свойств целостности и целесообразности модели системы.
Скажите, знакомо ли вам объектно-ориентированное программирование?



Ответ: ООП (Объектно-Ориентированное Программирование) стало неотъемлемой частью разработки многих современных проектов, но, несмотря на популярность, эта парадигма является далеко не единственной.

Давайте условимся на этой лекции говорить только об информационных системах. Прежде всего стоит ответить, зачем? Объектно-ориентированная идеология разрабатывалась как попытка связать поведение сущности с её данными и спроецировать объекты реального мира и бизнес-процессов в программный код.

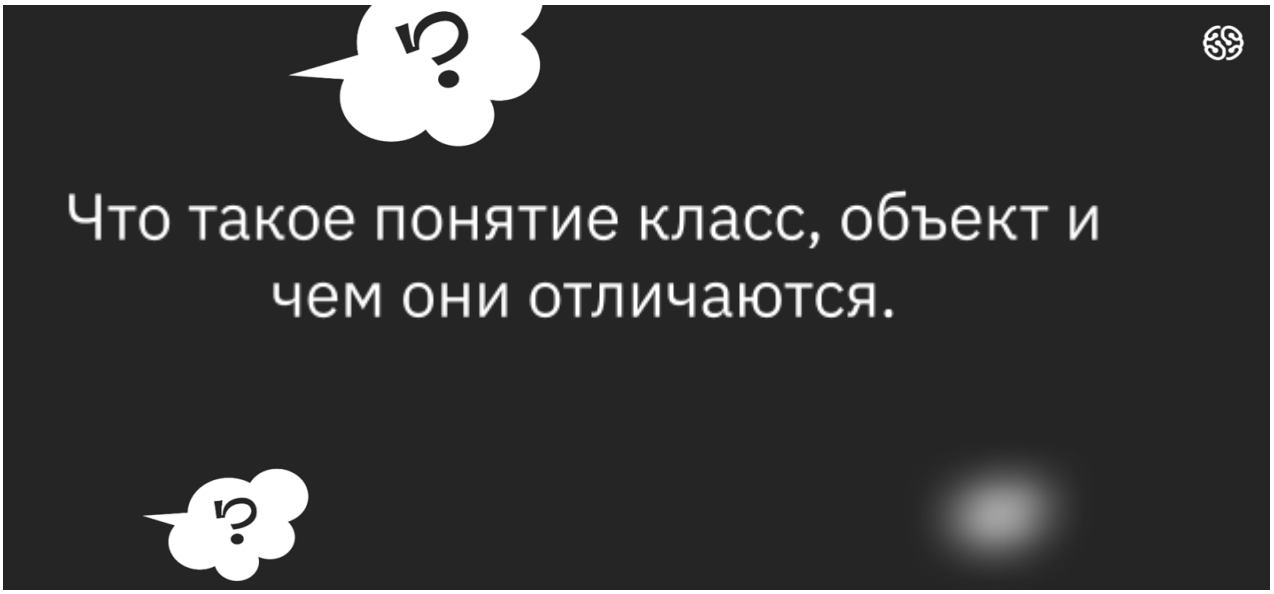
Задумывалось, что такой код проще читать и понимать человеком, т. к. людям свойственно воспринимать окружающий мир как множество взаимодействующих между собой объектов, поддающихся определенной классификации. Удалось ли идеологам достичь цели, однозначно ответить сложно, но де-факто мы имеем массу проектов, в которых требуется применение объектно-ориентированного подхода к анализу и разработке.

Если вам требуется написать одноразовый скрипт, который не нуждается в последующей поддержке, то и объектно-ориентированный подход в этой задаче, вероятнее всего, не пригодится. Однако, значительную часть жизненного цикла большинства современных проектов составляют именно поддержка и расширение. Само по себе наличие объектно-ориентированный подход не делает реализационную архитектуру безупречной, и может наоборот привести к излишним усложнениям.

> 💡 Объектно-ориентированное программирование стало нормой промышленной разработки информационных систем, а структурный анализ результатом которого выступает построение объектной модели (англ. ORM – object relative model) стало нормой для основного числа проектов для различных сфер применения.

Классы и объекты

Постарайтесь вспомнить или придумать определения понятиям класс, объект и объяснить чем они отличаются.



Ответ: Самое простое объяснение, класс — это описание каким должен быть предмет, а конкретный предмет, выступающий экземпляром класса – объект. Например, модель автомобиля достаточно четко определяет ожидаемые свойства автомобиля, каждый автомобиль определенной модели, построенный заданному по шаблону – экземпляр класса.

Таким образом, класс — это описание того, какими свойствами и поведением будет обладать объект. А объект — это экземпляр с собственным состоянием этих свойств. Мы уже сталкивались с классами, когда изучали анализ модели предметной области.

По отношению к информационным системам классы и объекты важны не как некие абстрактные понятия, а как основа основного подхода к разработке приложений – объектно ориентированному программированию.

Когда мы говорим «свойства и поведение», с точки зрения объектно-ориентированного программирования мы имеем в виду переменные и функции. На самом деле «свойства» — это такие же обычные переменные, просто они являются атрибутами какого-то объекта (их называют полями объекта). Аналогично «поведение» — это функции объекта (их называют методами), которые тоже являются атрибутами объекта. Разница между методом объекта и обычной функцией лишь в том, что метод имеет доступ к собственному состоянию через поля.

Относительно примера с автомобилем, свойством, или как еще называют атрибутом класса - будет число колес, а значением атрибутом экземпляра класса будет число

колес в конкретный момент времени от 0, на станции техобслуживания, до 4, при нормальной эксплуатации. Методом же будет называться поведение, к примеру изменение числа колес, так для класса автомобиль должен быть определен метод снятия и установки колеса.

Основные понятия и принципы объектно-ориентированного программирования

С точки зрения системой аналитики – программирование выходит за рамки проектирования системы, но так как программирование в информационных системах является неотъемлемой частью их реализации, системный аналитик должен иметь общие представления об основных концептах объектно-ориентированного программирования. Если заглянуть в программный код, то мы увидим несколько типовых конструкций

1. **new** — это ключевое слово, которое необходимо использовать для создания нового экземпляра какого-либо класса. В этот момент создается объект и вызывается конструктор.
2. **this (иногда self)** — это специальная локальная конструкция переменная (внутри методов), которая позволяет объекту обращаться из своих методов к собственным атрибутам. Так обращение
3. **constructor (construct, init или совпадает с именем класса)** — это специальный метод, который автоматически вызывается при создании каждого экземпляра объекта. Конструктор может принимать любые аргументы, как и любой другой метод. В каждом языке конструктор обозначается своим именем. Для рассматриваемого класса автомобиль, должен быть вызван этот метод для установки колес на штатные места до начала эксплуатации автомобиля, иначе использовать автомобиль не получится.

Классы могут обладать методами, которым не нужно состояние и, как следствие, создание объекта. В этом случае метод делают **статическим**. Для рассматриваемого примера такой метод будет похож на сигнал клаксона. Ведь его

реализация никак не учитывает окружающую среду, он просто звучит, нравится это кому или нет.

Существуют несколько основных принципов объектно-ориентированного программирования.

Эти основные принципы включены в аббревиатуру **SOLID** — это пять основных принципов проектирования в объектно-ориентированном программировании — Single responsibility, Open-closed, Liskov substitution, Interface segregation и Dependency inversion. В переводе на русский: принципы единственной ответственности, открытости / закрытости, подстановки Барбары Лисков, разделения интерфейса и инверсии зависимостей).

Принцип единственной ответственности - Первый принцип SOLID. С ним вы, наверняка, уже знакомы из других парадигм: «одна функция должна выполнять только одно законченное действие». Этот принцип справедлив и для классов: «Один класс должен отвечать за какую-то одну задачу». К сожалению с классами сложнее определить грань, которую нужно пересечь, чтобы принцип нарушался.

> 💡 Существуют попытки формализовать данный принцип с помощью описания назначения класса одним предложением без союзов, но это очень спорная методика, поэтому доверьтесь своей интуиции и не бросайтесь в крайности. Не нужно делать из класса швейцарский нож, но и плодить множество классов с однотипными методами – не соответствует принципам системного анализа

Традиционно в полях объекта могут храниться не только обычные переменные стандартных типов, но и другие объекты.

Ассоциация – в практике это отношение между вложенными в поля класса объекты. Например, для автомобиля (Car) можно определить колеса как отдельные объекты класса колесо (Wheel), для которого можно определить методы снять, поставить (dismount, mount).

Выглядит это как создание (new) экземпляра класса автомобиль, для которого инициализируются (costructor) экземпляры объектов класса колесо, всего 4 колеса (front_left_wheel, front_right_wheel, rear_left_whee, rear_right_wheel).
Для того чтобы снять переднее колесо вызывается метод снятия:

this.front_left_wheel.dismount()

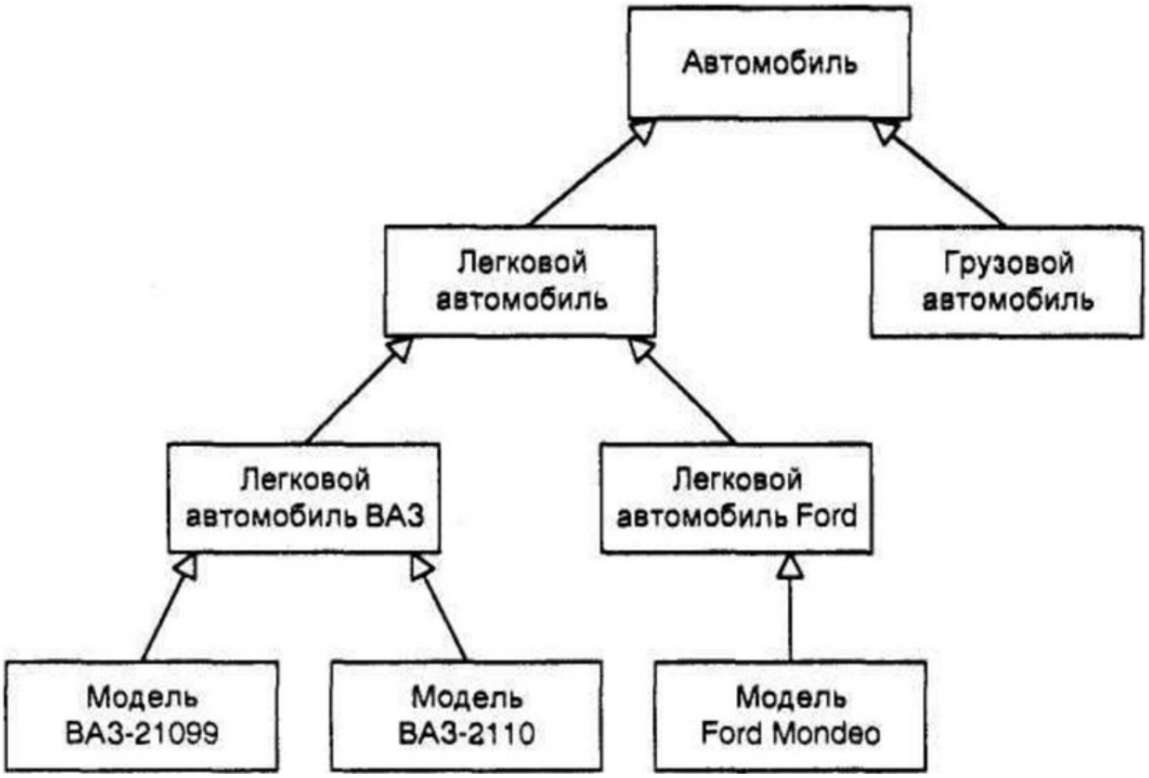
Отношение А эти объекты могут в свою очередь хранить какие-то другие объекты и так далее, образуя дерево (иногда граф) объектов. Это отношение называется ассоциацией.

Как мы ранее говорили, выделяют 2 типа ассоциации:

Композиция — случай, когда колесо утилизируется вместе с машиной и не может быть отчуждено от неё. Другими словами, жизненный цикл дочернего объекта совпадает с жизненным циклом родительского.

Агрегация — случай, когда колесо может быть использовано отдельно от машины.
Важнейшими принципами объектно-ориентированного программирования выступает **инкапсуляция, полиморфизм и наследование.** Разберем их предметно.

Наследование — это механизм системы, который позволяет, как бы парадоксально это не звучало, наследовать одними классами свойства и поведение других классов для дальнейшего расширения или модификации.



Что если, мы не хотим изготавливать одинаковые машины, а используем общую платформу, но с разным обвесом? ООП позволяет нам разделить логики на сходства и различия с последующим выносом сходств в родительский класс, а различий в

классы-потомки. Как это выглядит? Допустим на заводе изготавливается платформа с одинаковой подвеской, колесной базой, но разными двигателями, допустим бензиновый, дизельный и электрической.

С точки зрения объектно-ориентированного программирования проектируется система наследования включающая общие характеристики и поведение, куда может быть включены такие свойства как цвет, колесная база, конструкция подвески. Для каждой модели автомобиля меняется силовой агрегат, причем двигатель и коробка передач, а так же динамические характеристики автомобиля будут специфичны для наследников единой платформы.

Переопределение, это нюанс реализации наследования. Если же в классе-потомке переопределить уже существующий метод в классе-родителе, то сработает перегрузка. Это позволяет не дополнять поведение родительского класса, а модифицировать. В момент вызова метода или обращения к полю объекта, поиск атрибута происходит от потомка к самому корню — родителю.

Множественное наследование, это такой вариант наследования, когда свойства и поведение наследуются от нескольких классов. Так бывает в автомобиле, когда производители используют детали с другого конвейера. И такие автомобили иногда ездят, но точно их должным образом не тестировали.

Полиморфизм, это свойство системы, позволяющее иметь множество реализаций одного интерфейса. Ничего не понятно. Допустим мы говорим об автомобилях. Разные модели могут иметь разную реализацию педали акселератора, так в ДВС используется инжектор для контроля количества топлива поступающего в двигатель, а в автомобиле с электрическим двигателем – жидкое топливо не применяется, при нажатии на педаль акселератора.

Очевидно, что для водителя способ управления не должен меняться. Таким образом, реализация поведения определяется в дочернем классе, а интерфейс в родительском.

Это позволяет добавлять новые реализации не меняя порядок взаимодействия с потребителем, и вы, садясь в новый автомобиль не обязаны сдавать на отдельные права, разве только это не принципиально для вас новый класс.

Следующий принцип, это **Инкапсуляция**, представляющий собой контроль доступа к полям и методам объекта. Под контролем доступа подразумевается не только

можно/нельзя, но и различные валидации, подгрузки, вычисления и прочее динамическое поведение.

Во многих языках частью инкапсуляции является сокрытие данных. Для этого существуют модификаторы доступа:

- public — к атрибуту может получить доступ любой желающий
- private — к атрибуту могут обращаться только методы данного класса
- protected — то же, что и private, только доступ получают и наследники класса, в том числе

Геттеры и сеттеры (ацессоры, англ. Accessors) — это методы, задача которых контролировать доступ к полям. Геттер считывает и возвращают значение поля, а сеттер — наоборот, принимает в качестве аргумента значение и записывает в поле.

Это дает возможность снабдить такие методы дополнительными обработками. Например, сеттер при записи значения в поле объекта, может проверить тип, или формато-логический контроль. В геттер же можно добавить, ленивую инициализацию или кэширование, если актуальное значение на самом деле лежит в базе данных.

> 💡 В некоторых языках программирования ацессоры принято маскировать под свойства, что делает доступ прозрачным для внешнего кода, который и не подозревает, что работает не с полем, а с методом, у которого под капотом выполняется SQL-запрос или чтение из файла. Так достигается абстракция и прозрачность, но может привести к серьезным проблемам в случае наличия дефектов или особенностей реализации данных теневых функций данных функций

Интерфейсы – декларативный контракт обязывающий всех участников поддерживать определенный протокол взаимодействия.

Задача интерфейса — снизить уровень зависимости сущностей друг от друга, добавив больше абстракции. Не во всех языках присутствует этот механизм, но в языках объектно-ориентированного программирования со статической типизацией без них было бы совсем худо. Выше мы рассматривали абстрактные классы, затрагивая тему контрактов, обязующих имплементировать какие-то абстрактные методы. Так вот интерфейс очень смахивает на абстрактный класс, но является не

классом, а просто пустышкой с перечислением абстрактных методов (без имплементации). Другими словами, интерфейс имеет.

У интерфейса двустороннее применение:

1. По одну сторону интерфейса — классы, имплементирующие данный интерфейс.
2. По другую сторону — потребители, которые используют этот интерфейс в качестве описания типа данных, с которым они (потребители) работают.

Например, если какой-то объект помимо основного поведения, может быть сериализован, то пускай он имплементирует интерфейс «Сериализуемый». А если объект можно клонировать, то пусть он имплементирует еще один интерфейс — «Клонируемый». И если у нас есть какой-то транспортный модуль, который передает объекты по сети, он будет принимать любые объекты, имплементирующие интерфейс «Сериализуемый».

Так например в автомобиле есть шина данных и все электронные устройства должны поддерживать протокол взаимодействия, то есть обеспечивать контракт интерфейса.

Другие актуальные принципы разработки о которых должен знать аналитик

Как мы сказали ранее, принципы SOLID воспринимаются основным принципами, позволяющим структурно реализовать информационную систему на уровне компонентов программного кода.

Принцип единой ответственности позволяет структурно взглянуть на реализацию проекта, разбить элементы программного кода по функциональному признаку и получить прозрачную реализацию, обеспечивающую не только качественную реализацию проекта, но и делающие реализацию модернизируемой.

Но в дополнение к принципам SOLID, есть и другие принципы, важные для качественной реализации. Обычно эти принципы являются частью соглашения разработчиков по процессу разработки.

Среди этих принципов можно выделить KISS, DRY, YAGNI

> 💡 KISS (акроним для «Keep it simple, stupid» — «Делай проще, тупее») — принцип проектирования, принятый в ВМС США в 1960 году.

Этот принцип лучше всего иллюстрируется историей, когда Кларенс Джонсон, ведущий инженер авиастроительной компании Lockheed Skunk Works, вручил команде инженеров-авиаконструкторов набор инструментов, поставив им условие: механик среднего уровня должен суметь отремонтировать реактивный самолёт, который они проектировали, в полевых условиях, и только с примитивными подручными инструментами.

Таким образом, KISS позволяет проектировать и разрабатывать пригодные для дальнейшего сопровождения системы, он защищает от излишне мудреной реализации, понятной лишь одному создателю.

Не менее важный принцип - DRY.

> 💡 DRY (англ. dry — сухой, сушить) — один из основополагающих принципов разработки. Он расшифровывается как Don’t repeat yourself — «не повторяйтесь».

Когда разработчик пишет программный код, он должен думать о том, как можно переиспользовать тот или иной фрагмент, что можно выделить в универсальную функцию или класс, сделать модулем. При этом речь не идёт о создании программных библиотек под каждую функцию ради формальной галочки. Речь идет про включение в библиотеку схожей логики обработки, которая встречается в нескольких местах, и которую, возможно, есть смысл вынести в функцию.

А если в нескольких местах определена одна и та же функция, то её можно вынести в общий модуль. Ну и, если в коде приложения часто используется один и тот же модуль, то стоит рассмотреть вопрос, о том, чтобы его выделить в отдельную библиотеку. А если одна библиотека используется различными сервисами, то возможно следует сделать её отчуждаемой или спроектировать общий

микросервис, реализующий её функции, но это зависит от степени использования и объема передаваемых данных.

Следующий принцип - YAGNI

> 💡 YAGNI - Расшифровывается как You ain't gonna need it («Вам это не понадобится») и говорит о том, что не нужно включать в проект реализацию излишних функций

Принцип пришёл из экстремального программирования. Согласно ему создавать какую-то функциональность следует только тогда, когда она действительно нужна.

Дело в том, что в рамках Agile-методологий нужно фокусироваться только на текущей итерации проекта. Работать на опережение, добавляя в проект больше функциональности, чем требуется в данный момент, это, как правило, не очень хорошая идея, учитывая, как быстро могут меняться планы.

Само соглашение по процессу разработки представляют собой письменные и устные договоренности, реализующие требования к производственному процессу и качеству программного кода.

Прежде всего, перед началом производства работ устанавливаются правила оформления кода и иных артефактов, которые являются обязательными для выполнения всеми участниками проекта.

Стоит учесть, что современная разработка информационных систем ведется в интегрированных средах разработки, которые с одной стороны накладывают свои требования, а с другой реализуют возможности.

> 💡 IDE (от англ. Integrated Development Environment, «интегрированная среда разработки») — это программа, в которой разработчики пишут, проверяют, тестируют и запускают код, а также ведут большие проекты. Она включает в себя сразу несколько инструментов: редактор для написания кода, сервисы для его проверки и запуска, расширения для решения дополнительных задач разработки.

Существуют десятки разных IDE. Их можно делить на группы по разным критериям.

По стоимости среды IDE делятся на:

● Открытые, то есть полностью бесплатные. Такие часто используют начинающие разработчики для своих частных проектов.

● Условно-бесплатные. Обычно их можно скачать, но за расширенные функции придётся доплачивать. Такими пользуются как разработчики в частном порядке, так и компании.

● Полностью платные, то есть требующие покупки лицензии или подписки. Такими чаще всего пользуются компании, хотя иногда разработчики покупают их и для себя, если ведут крупные личные проекты.

По универсальности среды для программирования (IDE) делятся на:

● Одноязычные. Поддерживают только один конкретный язык программирования и оптимизированы именно для него.

● Мультиязычные. Поддерживают, конечно, не все, но многие языки программирования. Дополнительные можно добавлять с помощью устанавливаемых модулей.

Таким образом, рекомендуется оформлять требования соглашения реализующих соответствующие правила хорошего стиля в виде конфигураций статических анализаторов для IDE. Такие анализаторы могут найти и исправить проблемный код (ошибки, опечатки, уязвимости) на самых ранних этапах разработки без дополнительных трудозатрат, что кратно повышает эффективность конечной разработки.

Первым критерием возможности принятия изменений в общее хранилище программного кода проекта (репозиторий) является соответствие установленным стилевым правилам. Не соответствующие правилам изменения однозначно отклоняются. Если в процессе работ обнаруживаются разногласия в принципах оформления, которые не были оговорены в правилах, то после обсуждения свод правил расширяется.

Так например, соглашение описывающее качество программного кода может содержать следующие дополнительные пункты:

- в структуре проекта не должны присутствовать неиспользуемые в процессах производства технические каталоги и файлы;
- должна обеспечиваться независимость разработки, развертывания и масштабирования компонентов;
- должна быть обеспечена простота интеграции и взаимодействия компонентов

- должна быть обеспечена устойчивость к отказам
- обработку исключений желательно производить как можно более частными способами
- должна быть обеспечена независимость развертывания компонентов, исполнение компонентов в отдельных потоках исполнения
- должна быть обеспечена слабая связность компонентов
- разные разработчики должны понимать структуру и код проекта
- должна быть обеспечена децентрализация контроля над языками и данными
- реализация модулей должна допускать возможность линейного горизонтального масштабирования системы
- и другие ...

Помимо перечисленных требований соглашения содержат прикладные требования к ведению кодовой базы и организации производственного процесса, включая правила наименования, взаимодействия элементов соответствия архитектурным шаблонам, правила работы с клиентскими данными и другие нюансы, важные для качества разработки.

В целом при реализации системы рекомендуется придерживаться концепций Чистой архитектуры (Clean Architecture). Функционирование и взаимодействие объектов не должно нарушать установленных архитектурных правил.

Clean Architecture состоит из нескольких принципов, которые помогают разработчикам создавать качественное и устойчивое программное обеспечение. К основным принципам Clean Architecture можно отнести разделение на уровни:

- Уровень представления (Presentation)
- Уровень приложения (Application)
- Уровень домена данных (Domain)
- Уровень инфраструктуры (Infrastructure)

Уровень представления отвечает за взаимодействие с пользователем и обработку запросов. Уровень приложения выполняет бизнес-логику и координирует работу между уровнями представления и домена. Уровень домена содержит бизнес-логику и компоненты, отвечающие за работу с данными в определенной предметной области. Уровень инфраструктуры отвечает за поддержку структур приложения и связь с внешними системами, реализуя API - программный интерфейс приложения.

Для каждого из этих уровней должна быть решена проблема зависимостей, так что в каждом уровне приложения (например, Presentation и Domain) важно тщательно контролировать зависимости между компонентами. Например, компоненты внутри

слоя Presentation, выполняющего представление данных, не должны зависеть от компонентов в слое Domain, обеспечивающие извлечение этих данных.

Один из способов достичь этого - использование инверсии зависимостей (Dependency Inversion Principle, DIP), к слову являющийся неотъемлемой частью SOLID.

> 💡 Принцип инверсии зависимостей (англ. dependency inversion principle, DIP) — важный принцип объектно-ориентированного программирования, используемый для уменьшения связанности в компьютерных программах.

Вместо того чтобы компоненты в верхнем слое зависели от компонентов в нижнем слое, управление зависимостями и обмен данными происходит через общий интерфейс. Этот интерфейс реализует уровень абстракции, позволяющий упростить добавление, удаление или изменение компонентов с минимальными изменениями внутри каждого слоя.

Следующим принципом Clean Architecture являются граничные интерфейсы

> 💡 Граничные интерфейсы (Boundary Interfaces) - это интерфейсы, которые разделяют используемые элементы на две области: внутри приложения и вне его. Они служат для определения, какие элементы способны перейти за границу приложения и какие нет.

Граничные интерфейсы играют важную роль в Clean Architecture, так как они определяют, как пользовательский интерфейс должен общаться с приложением. Пользовательский интерфейс является внешней частью приложения, которая взаимодействует с пользователем и передает запросы в приложение. Граничные интерфейсы определяют, какие запросы пользовательский интерфейс может отправлять в приложение, и какие данные он может получать в ответ.

Примером граничного интерфейса является REST API веб-сервера, который позволяет клиентской стороне общаться с серверной стороной приложения. Если слой приложения использует граничные интерфейсы, то он был бы независим от клиентской стороны и мог бы быть более гибким при изменениях в клиентской стороне.

Использование принципов Clean Architecture сопряжено с риском ошибок, приводящих к снижению эффективности реализации, потере контекста и
Одной из частых ошибок при использовании Clean Architecture является неправильное определение границ приложения. Это может привести к тому, что различные уровни архитектуры могут переплетаться, что усложняет понимание и поддержку кода в будущем. Например, некоторые разработчики могут смешивать представление с бизнес-логикой, подрывая основные принципы Clean Architecture.

> 💡 Чтобы избежать ошибки смешения уровней архитектуры всегда необходимо определять границы между уровнями архитектуры и придерживаться их при проектировании и написании кода.

Еще одним распространенным моментом при работе с Clean Architecture является неправильная реализация Use Case. Use Case - это часть бизнес-логики приложения, которая определяет конкретный сценарий использования. Ошибка состоит в том, что разработчики часто пытаются комбинировать несколько Use Case вместе, порождая тем самым сложные функции связывающие различные бизнес процессы, что порождает сложности с развитием и масштабированием системы. Чтобы избежать этой проблемы, необходимо разбивать Use Case на более мелкие задачи и реализовывать их по отдельности.

Кроме того, неправильное использование зависимостей может пагубно сказаться на функционировании приложения. Различные компоненты должны быть независимыми друг от друга, что обеспечивает гибкость и легкость тестирования. Однако, когда разработчики пытаются сохранить связи между компонентами, это может привести к тому, что изменение одной части кода влияет на другую. Чтобы избежать этой проблемы, все компоненты должны быть разделены на отдельные модули, и зависимости должны быть определены явно.

> 💡 В целом можно сказать, что компоненты разрабатываемой системы должны иметь низкую связность и высокую согласованность.

Отсюда вытекают требования по внутренней структуре кодовой базы, подразумевающие что, разбиение на пакеты должно соответствовать следующим принципам: на верхнем уровне пакеты делятся по слоям архитектуры (repository, services, domain и т. п.). На следующем уровне — по отношению к совокупности бизнес-функций. Наборы более или менее универсальных вспомогательных (утилитарных) классов и методов сохраняются в отдельных пакетах, размещаемых

внутри области применения. Дерево пакетов, относящихся к клиентской части, должно иметь отдельный корень.

Ещё, в качестве примера важного принципа включаемого в соглашение можно выделить такой, что не следует использовать в контроллерах обработки логику, хранящую временное состояние (например сессионные данные пользователя). Следует помнить, что даже в рамках одной пользовательской сессии при использовании асинхронных запросов возможно возникновение феномена race condition.

> 💡 Состояние гонки (англ. race condition), также конкуренция — ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода.

Своё название ошибка получила от похожей ошибки проектирования электронных схем. Из-за неконтролируемого доступа к общей памяти состояние гонки может приводить к совершенно различным ошибкам, которые могут проявляться в непредсказуемые моменты времени, а попытка повторения ошибки в целях отладки со схожими условиями работы может оказаться безуспешной.

Основными последствиями могут быть:

- утечки памяти,
- ошибки сегментирования,
- порча данных,
- уязвимости,
- взаимные блокировки,
- утечки других ресурсов.
-

Таким образом, соглашение по разработке прежде всего призвано защитить систему от типовых стандартных ошибок, придать реализации проекта единый структурный вид, однозначно воспринимаемый всеми участниками команды разработки.

Кроме качества кода соглашение по разработке включает в себя описание модели разработки с точки зрения доставки функциональности.

Как правило, разработка ведется относительно короткими периодами (спринтами). В начале каждого периода определяется релизный объем — набор

функциональностей, который должен быть выполнен на момент завершения периода. В период реализации очередного релизного объема изменение его состава не допускается. Каждый период помимо реализации запланированной функциональности включает её тестирование. По завершению периода реализованная функциональность развертывается в продуктовой среде (или иной, по согласованию с заказчиком). Обычно, в целом не запрещается параллельная реализация более одного блока функциональности, но только при соответствующей координации работ, особенно с точки зрения их взаимной технической зависимости. Однако тестирование и развертывание релизных объемов должно выполняться последовательно.

Ещё соглашение по разработке может описывать порядок документирования. Для каждой отдельной функциональности в процессе (частично — до начала процесса) её реализации должен быть разработан объем документации, являющийся расширением и конкретизацией требований.

В зависимости от типа функциональности набор документации может включать сценарии использования (use case), сценарии тестирования, описание используемых нестандартных алгоритмов, интерфейсы взаимодействия и т. п. В общем случае набор и состав документации определяется реализующим функциональность. Но обязательным требованием к документации является возможность по ней, во-первых, другим участникам процесса прозрачно понять и утилизировать соответствующую функциональность, а, во-вторых, по совокупности всей документации выполнить приемо-сдаточные испытания системы.

Выполненная документация в структурированном виде размещается на ресурсах, доступных всем участникам процесса.

В целом соглашение по разработке структурно описывает весь процесс разработки и помогает команде разработке договориться и установить общие принципы реализации проекта.

Стоит сказать, что системный аналитик является активным участником команды разработки чьё влияние на реализацию проекта сложно переоценить и его погруженность в особенности рабочего процесса и соглашения о разработке окупается адекватной реализацией проектируемой системы.

Заключение

Мы рассмотрели основные вопросы структурного проектирования и познакомились с принципами объектно-ориентированного программирования. Узнали о составе соглашения по разработке, учитывающего принципы хорошего стиля, такие как SOLID, KISS, DRY, YAGNI.

Стоит отметить, что данные принципы могут и должны применяться системными аналитиками при проектировании систем.

На практическом занятии мы закрепим знания построив соответствующие схемы диаграммы классов и объектов в нотациях UML.

Владея основами объектно-ориентированного программирования аналитик сможет лучше понять проблемы реализации системы и точнее поставить задачу на разработку.

Что можно почитать еще?

1. Для более глубокого изучения принципов SOLID, о которых должен знать каждый разработчик https://habr.com/ru/companies/ruvds/articles/426413/
2. Принципы для разработки: KISS, DRY, YAGNI, BDUF, SOLID, APO и бритва Оккама https://habr.com/ru/companies/itelma/articles/546372/
3. Это классика, это знать надо: DRY, KISS, SOLID, YAGNI и другие полезные сокращения https://skillbox.ru/media/code/eto-klassika-eto-znat-nado-dry-kiss-solid-yagni-i-drugie-poleznye-sokrashcheniya/?ysclid=li3f2c2ick470052176

Используемая литература

1. Гради Буч, Джеймс Рамбо, Ивар Якобсон, «Язык UML. Руководство пользователя»
2. Бертран Мейер, «Почувствуй класс. Учимся программировать хорошо с объектами и контрактами»
3. И.Ю. Коцюба, А.В. Чунаев, А.Н. Шиков, «Основы проектирования информационных систем»