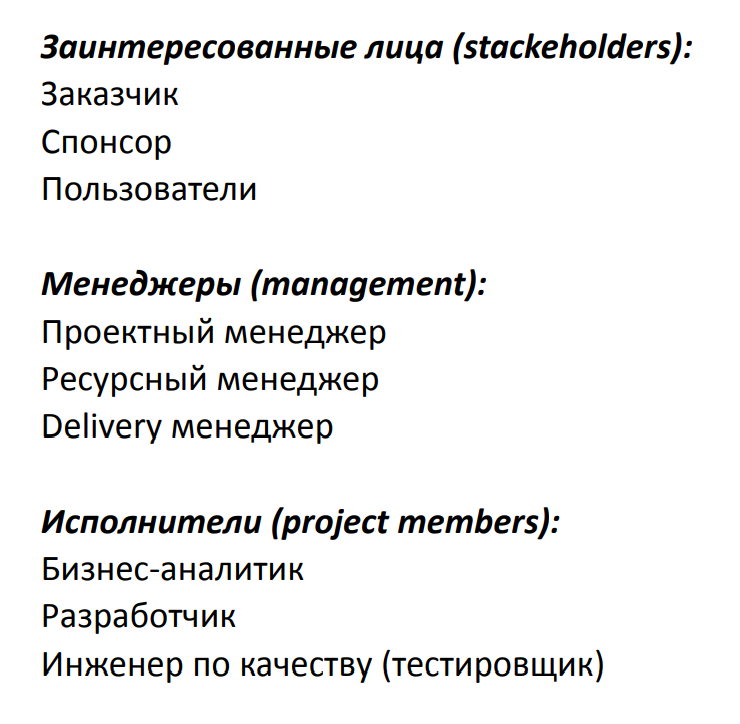
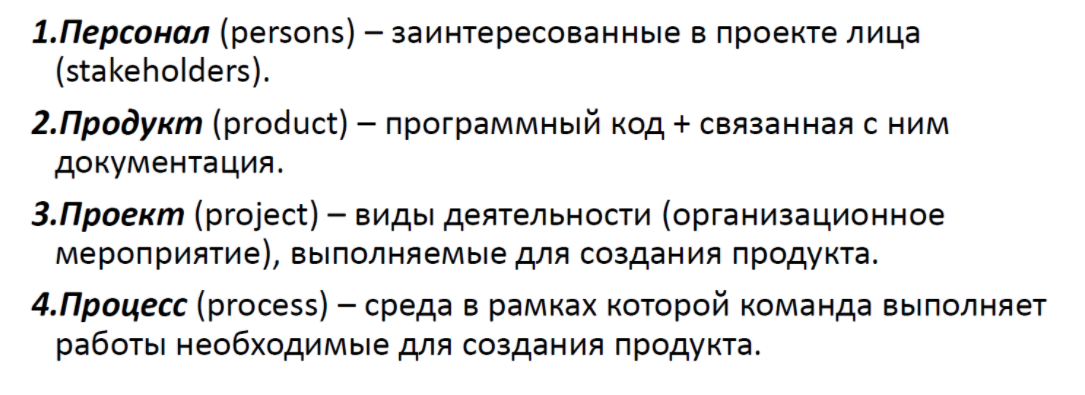
# https://habr.com/ru/post/210288/

# 

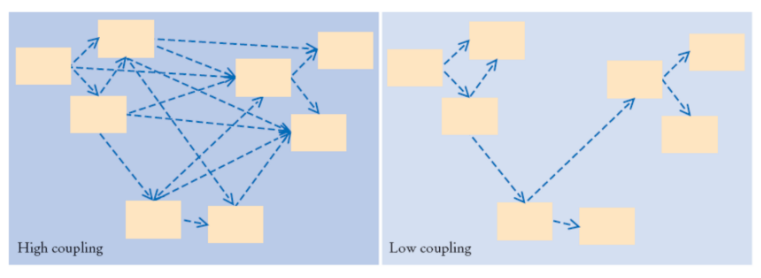
# 

# 1. Команда проекта (роли)





# 2. Низкое связывание и высокое зацепление



# **Низкое связывание**

**«Степень связности»** — мера неотрывности элемента от других элементов (либо мера данных, имеющихся у него о них).

**«Слабая» связность** — *распределение ответственностей и данных*, обеспечивающее *взаимную независимость классов*.

*Класс со «слабой» связностью:*

• Не зависит от внешних изменений;

• *Прост для повторного использования*.

• Обеспечить низкую связанность при создании экземпляра класса и связывании его с другим классом. ->

• Распределить обязанности между объектами так, чтобы степень связанности оставалась низкой

**Высокое зацепление**

Предметные области следует разделять по классам.

**Зацепление класса** — мера **подобия предметных областей** его методов:

• **«Высокое»** зацепление — сфокусированные подсистемы (предметная область определена, управляема и понятна);

• **«Низкое»** зацепление — абстрактные подсистемы. Затруднены:

• Восприятие;

• Повторное использование;

• Поддержка;

• Устойчивость к внешним изменениям.

• Необходимо обеспечить выполнение объектами разнородных функций. ->

• Обеспечить распределение обязанностей с высоким зацеплением.

*Преимущества*

• Классы с высокой степенью зацепления **просты в поддержке** и **повторном использовании.**

*Недостатки*

• Иногда бывает неоправданно использовать высокое зацепление для распределенных серверных объектов. В этом случае для обеспечения быстродействия необходимо создать несколько более крупных серверных объектов со слабым зацеплением. (??????)

# 3. Шаблоны GRASP

**GRASP** (general responsibility assignment software patterns — общие шаблоны распределения ответственностей) — шаблоны, используемые в объектно-ориентированном проектировании для решения общих задач по назначению ответственностей классам и объектам.

**Шаблоны «G.R.A.S.P.»** — хорошо документированные, стандартизированные и проверенные временем принципы объектно-ориентированного анализа, а не попытка привнести что-то принципиально новое.

*принципы*

* **Низкое связывание (Low Coupling)**(слабое взаимодействие(не зависит от внешних изменений)) **(мера сосредоточения на определённой задачи)**
* **Высокое зацепление (High Cohesion)(больше логики)**

–— должны взаим с друг другом, но уменьшить взаимодействие

* **Полиморфизм (Polymorphysm)** (должны повторно использовать код посредством интерфейса)

*роли* (распределили роли для понижения связности)

* **Информационный эксперт (Information Expert)** (model из MVC)
* **Создатель (Creator)** (класс для создания других объектов (абстр ф))
* **Контроллер**
* **Чистая Фабрикация (Pure Fabrication)(чистая выдумка)**
* **Посредник (Inderector)** (уменьшить зацепление)(выработать интерфейс)

*свойство*

* **Устойчивость к изменениям (Protected Variations)**

**1) Устойчивость к изменениям**

**(при измении одной части программы не меняются другие ( ну или это контролируемые изменения)(не ломается программа))**

**2.1) Низкое связывание (Low Coupling)**

-//-

**2.2) Высокое зацепление**

-//-

**2.3) Полиморфизм**

**(повторно использовать посредством интерфейсов)**

* Обязанности распределяются для различных вариантов поведения с помощью полиморфных операций для этого класса.
* Каждая внешняя система имеет свой интерфейс

**Критика:**

* **Преимущества:**
  + Впоследствии легко расширять и модернизировать систему.
* **Недостатки:**
  + Не следует злоупотреблять добавлением интерфейсов с применением принципа полиморфизма с целью обеспечения дееспособности системы в неизвестных заранее новых ситуациях.

**3.1) Информационный эксперт**

Ответственность должна быть назначена тому, кто владеет максимумом необходимой информации для исполнения **— информационному эксперту**

**Информационным экспертом может быть не один класс, а несколько**.

**3.2) Создатель**

Класс должен создавать экземпляры тех классов, которые он может:

* Содержать или агрегировать
* Записывать
* Использовать
* Инициализировать, имея нужные данные

**Пример:**

* необходимо определить, какой объект должен отвечать за создание экземпляра "ТоварПродажа".
* Логично, чтобы это был объект "Продажа", поскольку он содержит (агрегирует) несколько обьектов "ТоварПродажа".
* **Преимущества:**
  + Не повышает связанности, поскольку созданный класс, как правило, виден только для класса - создателя.

**Контроллер**

* **Отвечает за операции, запросы на которые приходят от пользователя, и может выполнять сценарии одного или нескольких вариантов использования (например, создание и удаление);**
* **Не выполняет работу самостоятельно, а делегирует компетентным исполнителям;**
* **Может представлять собой:**
  + **Систему в целом;**
  + **Подсистему;**
  + **Корневой объект;**
  + **Устройство.**
* **Преимущества:**
  + Удобно накапливать информацию о системных событиях (в случае, если системные операции выполняются в некоторой определенной последовательности).
  + Улучшаются условия для повторного использования компонентов (системные события обрабатываются Контроллером а не элементами интерфейса пользователя).
* **Недостатки:**
  + Контроллер может оказаться перегружен.

**Чистая фабрикация**

-//-

**Посредник**

**Слабая связность между элементами системы (и возможность повторного использования) обеспечивается назначением промежуточного объекта их посредником.**

**• Как перераспределить обязанности обьектов, чтобы обеспечить отсутствие прямого связывания?**

**• Присвоить обязанности по обеспечению связи между службами или компонентами промежуточному объекту.**

# 4. Виды шаблонов GoF (Gang of Four)

паттерны проектирования - это **концепция** (структура, схема) или **пример решения** той или иной **проблемы**, которое нужно будет подстроить под нужды вашей программы.

паттерн != алгоритм, если алгоритм - это чёткие шаги, то **паттерн - общая схема**

## **Порождающие (Creational)**

Заточены под **гибкое создание объектов** **без** внесения в программу лишних **зависимостей.**

Factory()

AbstractFactory()

Builder()

Singleton()

**Структурные (Structural)**

Показывают различные **способы построения связей между объектами.**

Структурные шаблоны проектирования упрощают проектирование путем выявления простого способа реализовать отношения между субъектами.

По итогу классы и объекты образуют более крупные структуры

Adapter()

Facade()

Most

Decorator

**Поведенческие (Behavioral)**

Заботятся об **эффективной коммуникации** между объектами.

Определяют алгоритмы и взаимодействие между классами и объектами, то есть их поведение

Следование этим шаблонам уменьшает связность системы и облегчает коммуникацию между объектами, что => улучшает гибкость программного продукта.

Посредник

Хранитель

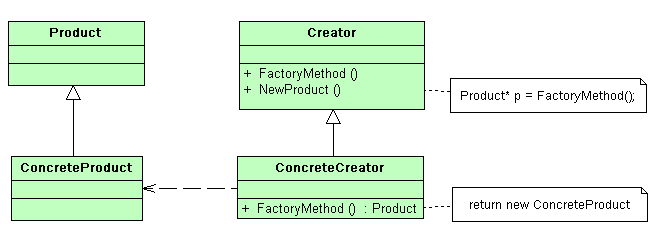
Наблюдатель

# 5. Шаблон Фабричный метод

**Фабричный метод** отделяет код производства продуктов от остального кода, который эти продукты использует. Благодаря этому, код производства можно расширять, не трогая основной.

Фабричный метод (**виртуальный конструктор**) — это порождающий (шаблон) паттерн проектирования, который **определяет общий интерфейс для создания объектов в суперклассе**, позволяя подклассам **изменять тип создаваемых объектов**. (путём имплементирования суперклассами интерфейса и переопределения методов (изменяя тип))

**Фабричный метод**, или **виртуальный конструктор**— [порождающий шаблон проектирования](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D0%BE%D0%B6%D0%B4%D0%B0%D1%8E%D1%89%D0%B8%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), предоставляющий [подклассам](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) [интерфейс](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Иными словами, данный **шаблон** [**делегирует**](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BB%D0%B5%D0%B3%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) **создание объектов наследникам родительского класса**. Это позволяет использовать в коде программы **не конкретные классы**, а **манипулировать абстрактными объектами на более высоком уровне.**



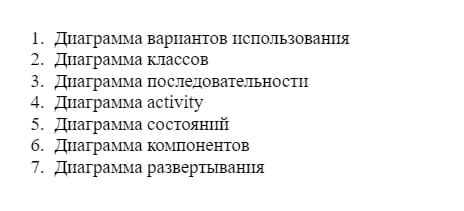
# 

# 

# 

# 

# 6. Основные диаграммы UML (перечислить и знать значение)



* Диаграмма вариантов использования (UseCase diagram) - описывает, какой функционал разрабатываемой программной системы доступен каждой группе пользователей.
* Диаграмма классов (Class diagram) - определяет типы классов системы и различного рода статические связи , которые существуют между ними
* Диаграмма деятельности (Activity diagram) - наглядно показывает, как поток управления переходит от одной деятельности к другой.

Позволяет более детально визуализировать конкретный случай использования. Это поведенческая диаграмма, которая иллюстрирует поток деятельности через систему.

* Диаграмма последовательности (Sequence diagram) - описывает отношения объектов в различных условиях

Они иллюстрируют, как различные части системы взаимодействуют друг с другом для выполнения функции, а также порядок, в котором происходит взаимодействие при выполнении конкретного случая использования.

* Диаграмма состояний (Statechart diagram) - показывает, как объект переходит из одного состояния в другое, а также события, которые влияют на эти переходы. Она помогает визуализировать весь жизненный цикл объектов и, таким образом, помогает лучше понять системы, основанные на состоянии.

# 7. Стандарты ТЗ

* ГОСТ 34.602-89

Если заказчик требует оформления документации в соответствии с государственным стандартом, (или это ТЗ для государственных проектов) выбор делается в сторону стандарта ГОСТ 34.602-89. Подготовка Технического задания по ГОСТ 34.602 -89 требует значительных временных затрат, тк очень много требований к оформлению.(целых 9 разделов)

* IEEE Std 830

Если поставлены сжатые сроки подготовки ТЗ и заказчик не требует оформления документации в соответствии с государственным стандартом, то можно использовать шаблон технического задания по стандарту IEEE Std 830. Стандарт IEEE Std 830 рекомендует различные способы структурирования детальных требований для классов систем, поскольку не существует оптимальной структуры для всех систем.

* Корпоративный шаблон

Существует и третья альтернатива для выбора шаблона Технического задания, когда заказчик предлагает использовать принятый в компании Корпоративный шаблон для описания требований к информационным системам, избрав приоритетные для себя пункты.

# 8. Структура ТЗ

**Техническое задание** – технический документ (спецификация), оговаривающий набор требований к системе и утверждённый как заказчиком/пользователем, так и исполнителем/производителем системы. Такая спецификация может содержать также системные требования и требования к тестированию.

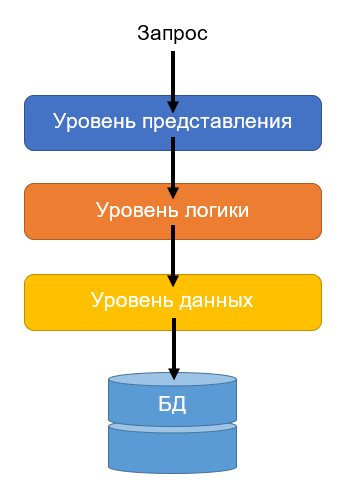
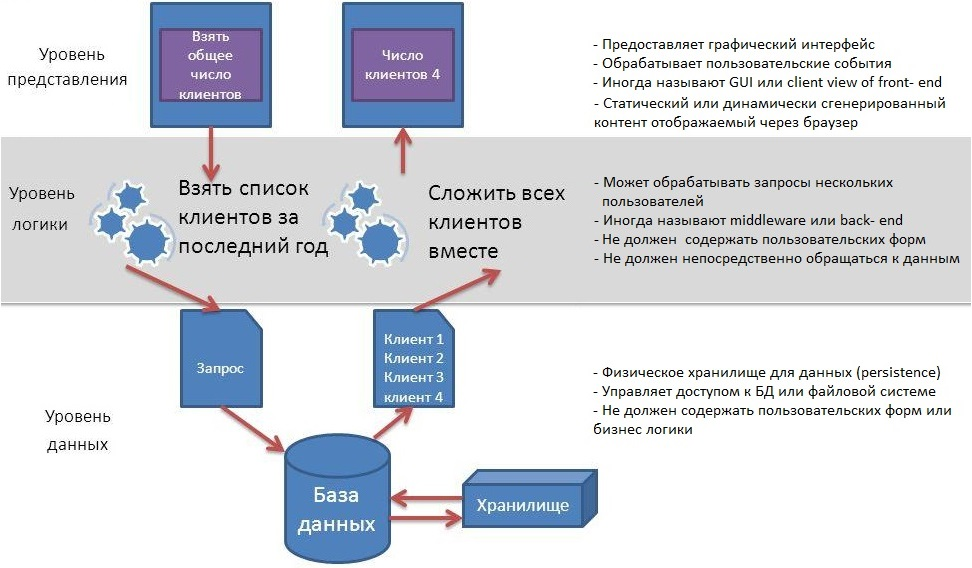
**Структура тз:**

* Общие сведения
* Назначение системы
* Требования к системе
  + Функциональные требования
    - Описание пользователей системы
    - Описание вариантов использования
  + Технические требования
    - Архитектура системы
    - Требования к пользовательскому интерфейсу
    - Требования надежности
* Порядок приема систем
  + Приемочные тесты
  + Сроки исполнения

# 9. Слои и уровни приложения

Приложение делится на уровни, каждый из которых взаимодействует лишь с двумя соседними. Поэтому запросы к БД, которая обычно располагается в самом конце цепочки взаимодействия, проходят последовательно сквозь каждый «слой».

Архитектура [не подразумевает](https://www.oreilly.com/ideas/software-architecture-patterns/page/2/layered-architecture) какое-то обязательное количество уровней — их может быть три, четыре, пять и больше. Чаще всего используют трехзвенные системы: с уровнем представления (клиентом), уровнем логики и уровнем данных.

Когда приложение делится на зависимые уровни, то каждый уровень, зависимый от уровня ниже называют слоем.

Слой – уровень зависимый от другого уровня. Отсюда следует, что уровень независимый от другого уровня, не считается слоем (на примере выше у нас таковым является уровень данных)

* уровни используют для понимания глубины слоя
* слои используют для понимания того, из каких частей состоит приложение

# 10. Шаблон DAO (Data Access Object)

Это **структурный шаблон**

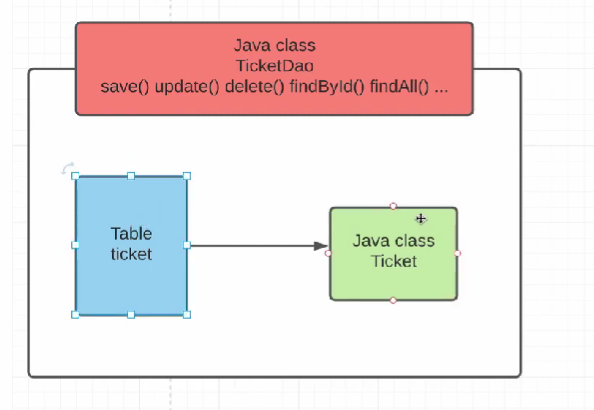
в виде **класса DAO**

между ***реляционными и объектными* моделями данных**

каждый объект представляет какую-то из таблиц

По существу, DAO **выполняет функцию адаптера между компонентом и источником данных.**

DAO полностью **скрывает детали реализации** *источника данных* **от клиентов**



**DAO для абстрагирования и инкапсулирования доступа к источнику данных. DAO управляет соединением с источником данных для получения и записи данных.**

DAO - **структурный шаблон**,реализует необходимый для работы с источником данных механизм доступа. Источником данных может быть персистентное хранилище, внешняя служба, репозиторий, или бизнес-служба, обращение к которой осуществляется при помощи протокола CORBA Internet Inter-ORB Protocol (IIOP) или низкоуровневых сокетов. Использующие DAO бизнес-компоненты работают с более простым интерфейсом, предоставляемым объектом DAO своим клиентам. DAO полностью скрывает детали реализации источника данных от клиентов. Поскольку при изменениях реализации источника данных представляемый DAO интерфейс не изменяется, этот паттерн дает возможность DAO принимать различные схемы хранилищ без влияния на клиенты или бизнес-компоненты. По существу, DAO **выполняет функцию адаптера между компонентом и источником данных.**

# 11. Что такое Сервисы?

Web-сервис (служба) – программа, которая организовывает взаимодействие между сайтами. Информация с одного портала передается на другой.

На сегодняшний день наибольшее распространение получили следующие протоколы реализации веб-сервисов:

SOAP (Simple Object Access Protocol) — по сути это тройка стандартов SOAP/WSDL/UDDI

REST (Representational State Transfer)

XML-RPC (XML Remote Procedure Call)

# 12. Шаблоны SOLID (перечислить и знать значение) (почитать самой)

S - принцип единственной ответственности

O - принцип открытости/закрытости

L - принцип подстановки Барбары Лисков

I - принцип разделения интерфейса

D - принцип инверсии зависимостей

Принцип единственной ответственности Формулировка: Существует лишь причина, приводящая к изменению класса. Каждый объект должен одну ответственность и эта ответственность должна быть полностью инкапсулирована в класс. Все его поведения должны быть направлены исключительно обеспечение этой ответственности.

Принцип открытости/закрытости Формулировка: Программные сущности должны быть (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для модификации. \* открыты для расширения: поведение сущности может быть расширено, путём создания новых типов сущностей. • закрыты для изменения: в результате расширения поведения сущности, не должны вноситься изменения в код, который эти сущности используют.

Принцип подстановки Барбары Лесков Объекты программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы. Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом (а также подтипы должны быть заменяемы базовыми типами). Поведение наследуемых классов не должно противоречить поведению, заданному базовым классом, то есть поведение наследуемых классов должно быть ожидаемым для кода, использующего переменную базового типа. принцип предупреждает разработчика о том, что изменение унаследованного производным типом поведения очень рискованно.

Принцип разделения интерфейсов Формулировка 1: клиенты не должны зависеть от методов, которые они не используют Формулировка 2: лучше иметь много маленьких интерфейсов, чем один большой. Как и в случае с другими принципами проектирования классов, мы пытаемся избавиться от ненужных зависимостей в коде, сделать код легко читаемым и легко изменяемым.

Принцип инверсии зависимости Формулировка:

• Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба должны зависеть от абстракции.

• Абстракции должны зависеть от деталей. Детали должны зависеть от абстракций.

# 

# 13. Виды тестирования (перечислить и знать значения)

**1 Модульные тесты** - модульные тесты работают на очень низком уровне, близко к исходному коду приложения. Они заключаются в тестировании отдельных методов и функций классов, компонентов или модулей, используемых в ПО. Модульные тесты, как правило, не требуют больших расходов на автоматизацию и могут выполняться сервером непрерывной интеграции очень быстро.

**2 Интеграционное тестирование** - в ходе интеграционного тестирования проверяется, хорошо ли работают вместе различные модули и сервисы, используемые приложением. Например, можно протестировать взаимодействие с базой данных или убедиться, что микросервисы работают вместе так, как задумано. Этот вид тестирования является более затратным, поскольку для проведения тестов требуется запуск различных компонентов приложения.

**3 Функциональные тесты -** В функциональных тестах основное внимание уделяется бизнес-требованиям к приложению. Они проверяют только результат некоторого действия и не проверяют промежуточные состояния системы при выполнении этого действия. Иногда возникает путаница между понятиями интеграционных и функциональных тестов, так как и те и другие требуют взаимодействия нескольких компонентов друг с другом. Разница в том, что интеграционный тест нужен просто чтобы убедиться, что вы можете отправлять запросы к базе данных, тогда как функциональный тест будет ожидать получения из базы данных определенного значения в соответствии с требованиями продукта.

**4 Сквозные тесты -** Сквозное тестирование копирует поведение пользователя при работе с ПО в контексте всего приложения. Оно обеспечивает контроль того, что различные схемы действий пользователя работают должным образом. Сценарии могут быть как очень простыми (загрузка веб-страницы или вход в систему), так и гораздо более сложными (проверка почтовых уведомлений, онлайн-платежей и т. д.). Сквозные тесты очень полезны, но их выполнение обходится довольно дорого, к тому же, когда они автоматизированы, такие тесты тяжело обслуживать. Рекомендуется иметь в наличии несколько основных сквозных тестов и активнее полагаться на более низкие уровни тестирования (модульные и интеграционные тесты), чтобы получать возможность быстро выявлять критические изменения.

**5 Приемочное тестирование -** Приемочные тесты — это формальные тесты, которые призваны проверить, отвечает ли система требованиям бизнеса. При этом необходим запуск всего приложения, и основное внимание уделяется воспроизведению поведения пользователей. В ходе этого тестирования возможен даже замер производительности системы, и в случае несоответствия установленным требованиям внесенные изменения могут быть отклонены.

**6 Тестирование производительности -** Тесты производительности проверяют поведение системы при нахождении под значительной нагрузкой. Эти тесты не являются функциональными и могут принимать различную форму, но служат они для оценки надежности, стабильности и доступности платформы. Например, это может быть наблюдение за временем отклика при выполнении большого количества запросов или наблюдение за тем, как система себя ведет со значительными объемами данных.

Тесты производительности по своей природе требуют серьезных затрат при реализации и выполнении, но именно они помогают понять, не снижают ли новые изменения производительность вашей системы.

**7 Smoke-тестирование -**Smoke-тесты представляют собой базовые тесты, которые проверяют основные функциональные возможности приложения. Они должны выполняться быстро, поскольку цель таких тестов — убедиться, что основные возможности системы работают как запланировано. Smoke-тесты полезно запускать сразу после создания новой сборки (для определения, можно ли запускать более ресурсоемкие тесты) или сразу после развертывания (чтобы убедиться, что приложение работает правильно в новой, только что развернутой среде).

# 14. Шаблон Фасад

**Шаблон фасад** — структурный шаблон проектирования, позволяющий скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

Проблема

Минимизировать зависимость подсистем некоторой сложной системы и обмен информацией между ними.(Клиенты хотят получить упрощенный интерфейс к общей функциональности сложной подсистемы.)

**Описание проблемы**

При проектировании сложных систем, зачастую применяется т.н. принцип декомпозиции, при котором сложная система разбивается на более мелкие и простые подсистемы. Благодаря такому подходу, отдельные компоненты системы могу быть разработаны изолированно, затем интегрированы вместе. Однако возникает проблема — высокая связность модулей системы. Это проявляется, в первую очередь, в большом объеме информации, которой модули обмениваются друг с другом.

Таким образом, минимизация зависимости подсистем, а также снижение объема передаваемой между ними информации — одна из основных задач проектирования.

Один из способов **решения данной задачи** — использование паттерна «Фасад».

Паттерн Facade предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Facade определяет интерфейс более высокого уровня, упрощающий использование подсистемы.Паттерн Facade "обертывает" сложную подсистему более простым интерфейсом

Фасад агрегирует классы, реализующие функциональность этой подсистемы, но не скрывает их. Важно понимать, что клиент, при этом, не лишается более низкоуровневого доступа к классам подсистемы, если ему это, конечно, необходимо.

# 15. Шаблон Адаптер

**Адаптер** — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

**Проблема**

Представьте, что вы делаете приложение для торговли на бирже. Ваше приложение скачивает биржевые котировки из нескольких источников в XML, а затем рисует красивые графики.

В какой-то момент вы решаете улучшить приложение, применив стороннюю библиотеку аналитики. Но вот беда — библиотека поддерживает только формат данных JSON, несовместимый с вашим приложением.

**Решение**

Вы можете создать *адаптер*. Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого. Например, вы можете обернуть объект, работающий в метрах, адаптером, который бы конвертировал данные в футы.

Адаптеры могут не только переводить данные из одного формата в другой, но и помогать объектам с разными интерфейсами работать сообща.

# 16. Rational Unified Process

**Унифицированный процесс Rational —** это универсальная методология распределения задач и сфер ответственности при разработке программного обеспечения. Её цель – создание высококачественного по, отвечающего потребностям и запросам пользователей.

**Данный подход описывает, кто делает что делает, как и когда.**

**В основе RUP лежит шесть главных принципов:**

* Итеративная модель разработки – **устранение рисков** на каждой стадии проекта позволяет лучше понять проблему и вносить необходимые изменения, пока не будет найдено приемлемое решение;
* **Управление требованиями** – RUP описывает процесс организации и отслеживания функциональных требований, документации и выбора оптимальных решений (*как в процессе разработки, так и при ведении бизнеса*);
* **Компонентная архитектура** – архитектура системы разбивается на компоненты, которые можно использовать как в текущем, так и в будущих проектах;
* **Визуальное моделирование ПО** – RUP методология разработки показывает, как создать визуальную модель программного обеспечения, чтобы понять структуру и поведение архитектуры и его компонентов;
* **Проверка качества ПО** – в процессе разработки программного обеспечения контролируется качество всех действий команды;
* **Контроль внесённых изменений** – отслеживание изменений позволяет выстроить непрерывный процесс разработки. Создаётся благоприятная обстановка, в рамках которой команда будет защищена от изменений в рабочем процессе.

**Работа над процессом** – команда разработчиков RUP тесно работает с покупателями, партнёрами и группами компаний, постоянно обновляя методологию.

**RUP оптимизирует командную работу** – обеспечивает команде разработчиков свободный доступ к базе знаний с инструкциями для использования программных средств. Это помогает быстрее справиться с критическими проблемами. Благодаря этому команда легко находит общий язык в процессе работы над проектом.

**RUP ориентирован на создание и поддержание моделей** – вместо написания большого количества бумажной документации при использовании **RUP** методологии создаются модели – семантические представления разрабатываемого командой программного обеспечения.

**RUP** **– это инструкция использования Unified Modelling Language (UML)** – **UML** позволяет команде легко донести свои требования к проекту, его архитектуру и план реализации.

**RUP – это конфигурируемый процесс**, он подходит как небольшим группам разработчиков, так и крупным организациям.

**В основе RUP лежит шесть главных принципов**:

* **Итеративная модель разработки** – устранение рисков на каждой стадии проекта позволяет лучше понять проблему и вносить необходимые изменения, пока не будет найдено приемлемое решение;
* **Управление требованиями** – **RUP** описывает процесс организации и отслеживания функциональных требований, документации и выбора оптимальных решений (*как в процессе разработки, так и при ведении бизнеса*);
* **Компонентная архитектура** – архитектура системы разбивается на компоненты, которые можно использовать как в текущем, так и в будущих проектах;
* **Визуальное моделирование ПО – RUP методология разработки показывает**, как создать визуальную модель программного обеспечения, чтобы понять структуру и поведение архитектуры и его компонентов;
* **Проверка качества ПО** – в процессе разработки программного обеспечения контролируется качество всех действий команды;
* **Контроль внесённых изменений** – отслеживание изменений позволяет выстроить непрерывный процесс разработки. Создаётся благоприятная обстановка, в рамках которой команда будет защищена от изменений в рабочем процессе.

Данный подход описывает, кто делает что делает, как и когда. **RUP** можно представить в виде четырёх основных **элементов**:

Работники – «Кто»

Элемент «*работник*» определяет поведение и ответственность всех членов команды, сконцентрированных на общей цели – создании искусственных объектов (*артефактов*). В **RUP работник** – это скорее **роль**, определяющая, как индивидуумы должны выполнять свою работу. Работник не только **производит определённые действия**, но также **владеет набором артефактов.**

Артефакты (искусственные объекты) – «Что»

В методологии **RUP** **объекты или информация, производимая и изменяемая** в процессе работы над финальным результатом. Артефакты служат как вводными данными для действий работников, так и результатами этих действий.

Действия – «Как»

Действие — это е**диница работы, которую может выполнить работник**. У каждого действия есть четко обозначенная цель. Каждое действие назначено определенному работнику. Действия могут включать в себя создание или обновление артефактов, таких как модель, класс или план.

Рабочий процесс – «Когда»

Рабочий процесс представляет собой **последовательность действий,** приводящих к видимому результату. В терминах **UML** можно представить рабочий процесс как диаграмму последовательности, диаграмму кооперации или диаграмму активности:

# 17. Гибкие методологии разработки ПО (общий смысл, перечислить)

XP, Scrum, Kanban, Agile

**XP** - экстремальное программирование (один учится, другой пишет приложение)

**Kanban** - как Trello - где есть четкий график выполнения задач, ограниченное количество задач выполняемых одновременно

**SCRUM** набор конкретных правил, говорящих, как именно организовывать работу

Agile - совмещение всех 3х предыдущих методологий

• Sprint – время, за которое создается инкремент продукта — готовый для конечного пользователя продукт.

**Благодаря Agile-методам, удается сделать процесс руководства проектами более упорядоченным.**

**Принципы гибкой разработки**

*Гибкие подходы основываются на следующих базовых принципах:*

1. Работающее ПО является основным средством измерения развития проекта.

2. В таком случае, для развития проекта требуется, чтобы ПО разрабатывалось и предоставлялось быстро путем добавления небольших улучшений.

3. Даже поздние изменения в требованиях должны быть приняты (модель разработки на основе небольших приращений функциональности помогает их принимать).

4. Личное общение является более предпочтительным, чем обмен документами.

5. Непрерывная обратная связь с потребителем и вовлечение потребителя в разработку является необходимым для создания высококачественного ПО.

6. Простое проектирование, которое развивается и улучшается со временем, является лучшим подходом в сравнении с выполнением заранее детального проектирования, которое учитывает все возможные сценарии использования создаваемого ПО.

7. Сроки передачи готового ПО определяются квалифицированными командами талантливых разработчиков (а не путем распоряжений).

# 18. Итеративные (инкрементарные) методологии разработки ПО

# 19. Жизненный цикл программного продукта

**Жизненный цикл программного обеспечения** (Software Life Cycle Model) — это **период времени**, который начинается **с момента принятия решения о создании программного продукта** и заканчивается в **момент его полного изъятия из эксплуатации**. Этот цикл — процесс построения и развития ПО.

Жизненный цикл можно представить в виде моделей. В настоящее время наиболее распространенными являются: ***каскадная*, *инкрементная*** (*поэтапная модель с промежуточным контролем*) и ***спиральная*** модели жизненного цикла.

—-------------------------------------------

**каскад (водопад)** – для простых проектов - каждая последующая стадия идёт только после завершения предыдущей

—-------------------------------------------------------------------------------

**инкрементарная** – заказчик знает, какой полноценный продукт он хочет, его разрабатывают, деплоят, оно работает, заказчик может сказать, что вот хочу ещё это добавить и того, и происходит такое себе “расширение приложения” (т е готовое полноценное, а потом постоянно обновляется)

+

* *Не нужно вкладывать много денег на начальном этапе*
* *Можно быстро получить фидбэк от пользователей и оперативно обновить техническое задание*
* *Ошибка обходится дешевле.*

-



—----------------------------------------------------------------------------------

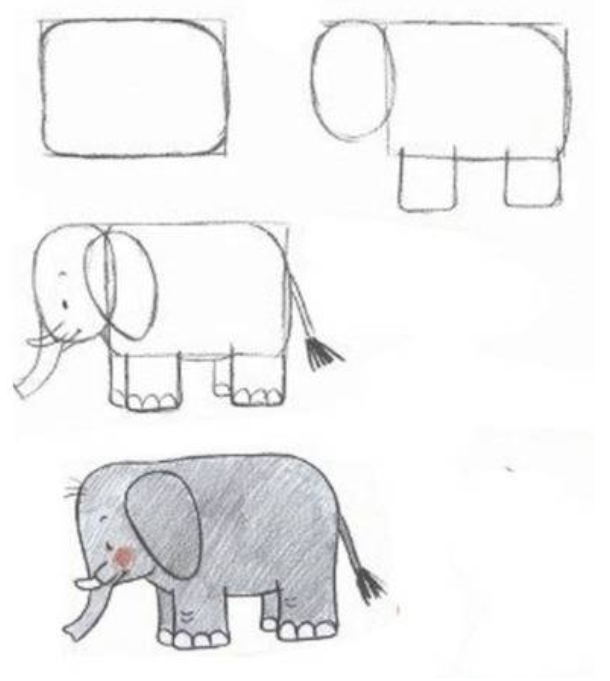
**итеративная** – заказчик хочет по чуть-чуть поэтапно проверять создаваемое приложение – вот создали набросок, где уже что-то работает (минимальный продукт) – заказчик проверил, задеплоили, приложение скачивают – заказчик такой - а давайте улучшим!

+

* *Постоянное тестирование пользователями*
* *Быстрый выпуск минимального продукта*

*-*

* *Отсутствие фиксированного бюджета и сроков*

**

*—------------------------------------------------------------------------------------------------------*

**спираль** – (*исследовательских проектах и там, где высоки риски.* )в начале разработчики хорошо обдумыват все риски, ведь каждая последующая часть разработки оосновывается на предыдущей

инкрементарность+водопад

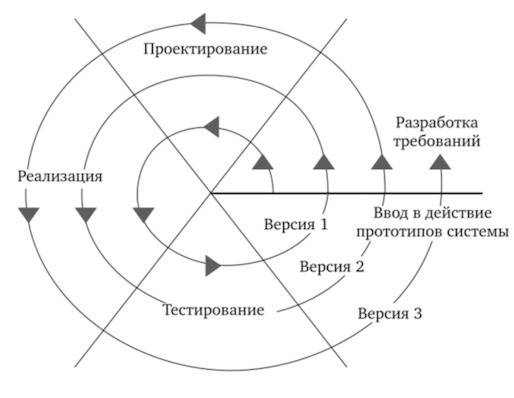
создали продукт на основе водопада - выкатили. потом заказчик решил дополнить его и опять идёт водопад

+

* *проработке рисков.*

-

* *можно застрять на начальном этапе*
* *долго и стоит дорого.*



# 20. Водопадная модель разработки ПО (стадии)

одна из наиболее широко используемых моделей (особенно в медицине и космических отраслях)(для рутинных проектов, где всё понятно как делать)



Плюсы:

* **Предсказуемая** (просто контролировать)
* **Старый проверенный** метод
* **стоимость** и **время** проекта определяется **в самом начале**

Минусы:

* Слегка **идеализированная** и не предполагает изменений требований(**не гибкая**) (**требования нельзя изменить** о окончания разработки)
* **Много технической документации** *– задерживает работы*
* Заказчик видит результат только в конце и может дать фидбек только после создания проекта

# 21. Диаграмма классов UML

Диаграмма классов – это UML-диаграмма, которая демонстрирует **общую структуру иерархии классов системы**, их **коопераций**, **атрибутов (полей)**, **методов**, **интерфейсов** и **взаимосвязей** между ними.

Диаграмма классов является ключевым элементом в объектно-ориентированном моделировании.

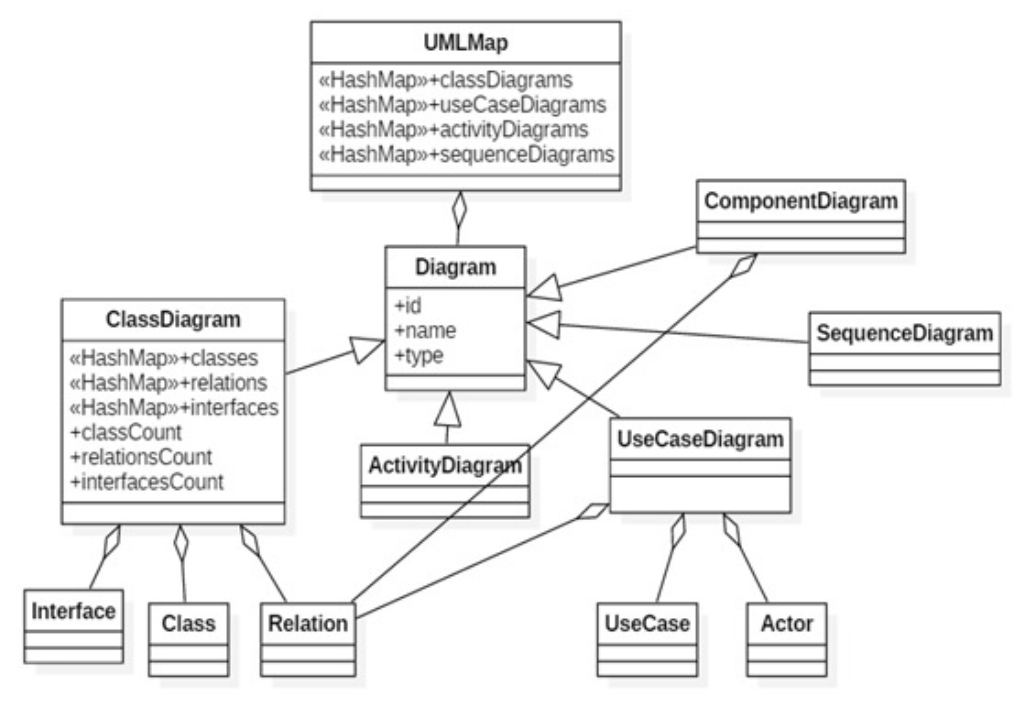
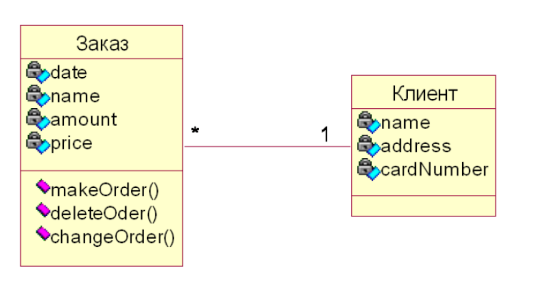
На диаграмме классы представлены в рамках, содержащих три компонента:

В верхней части написано имя класса.

Если класс абстрактный — то его имя пишется полужирным курсивом.

Посередине располагаются поля (атрибуты) класса.

Нижняя часть содержит методы класса.



Отношения между классами смотри в вопросе 24

# 22. Диаграмма вариантов использования

Данная диаграмма состоит из актеров, вариантов использования (прецедентов) и отношений между ними. Суть данной диаграммы состоит в следующем: проектируемая система представляется в виде множества актеров, взаимодействующих с системой с помощью так называемых вариантов использования. При этом ***актером*** (действующим лицом) называется любой объект, субъект или система, взаимодействующая с моделируемой системой извне. В свою очередь каждый ***вариант использования*** определяет некоторый набор действий, совершаемых системой при взаимодействии с актером. Графически ***актера*** можно определить в виде “проволочного человечка”. ***Вариант использования*** обозначается на диаграмме эллипсом, внутри которого содержится его описание, обозначающее выполнение какой-либо операции или действия. Связи между актерами и вариантами отображаются с использованием ***отношений*** четырех видов:

- ассоциаций;

- обобщения;

- включения (зависимость со стереотипом «include»);

- расширения (зависимость со стереотипом «extend»).

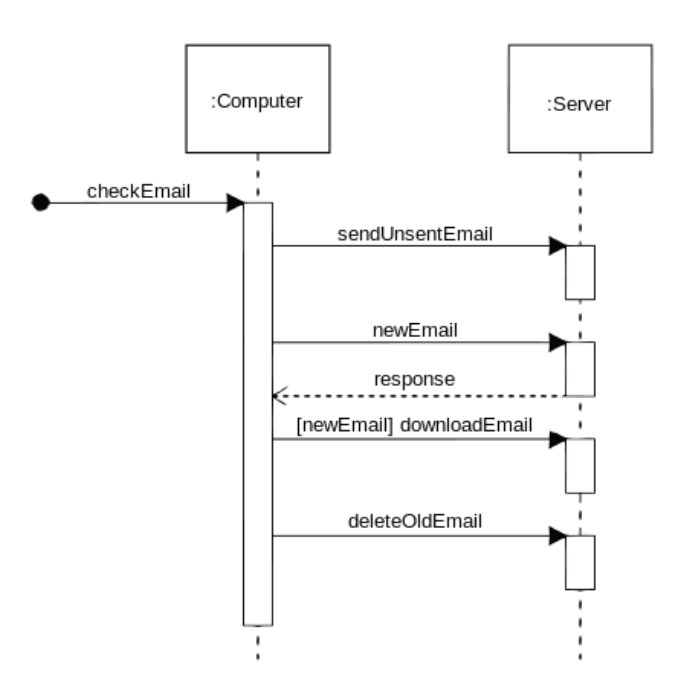
***Отношение ассоциации*** служит для обозначения взаимодействия актера с вариантом использования. Ассоциация отображается в виде сплошной линии.

***Отношение включения*** указывает, что некоторое заданное поведение одного варианта использования *обязательно* включается в качестве составного компонента в последовательность поведения другого варианта использования. Стрелка включения (пунктирная) должна быть направлена от базового (составного) варианта к включаемому и помечена стереотипом «include».

В отличие от отношения включения, ***отношение расширения*** определяет *потенциальную возможность* включения поведения одного варианта использования в состав другого. Т. е. дочерний вариант использования может как вызываться, так и не вызываться родительским. Стрелка расширения должна быть направлена от включаемого варианта к базовому и помечена стереотипом «extend».

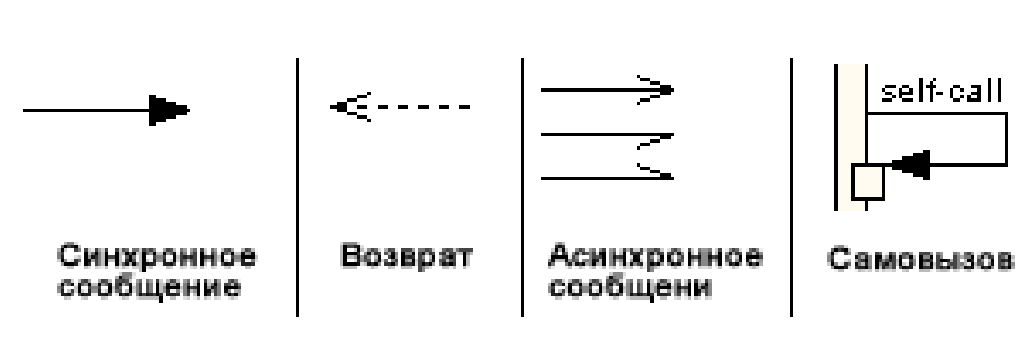
***Отношение обобщения*** служит для указания того факта, что некоторая сущность А может быть обобщена до сущности В. В этом случае сущность А будет являться специализацией сущности В. На диаграмме данный вид отношения можно отображать только между однотипными сущностями (между двумя вариантами использования или двумя актерами). Графически данное отношение обозначается сплошной линией со стрелкой, в виде незакрашенного треугольника, от потомка к родителю.

# 23. Диаграмма последовательностей

**Диаграмма последовательностей** – отображает последовательность, в которой объекты в процессе взаимодействия обмениваются сообщениями.

**Типы сообщений:**

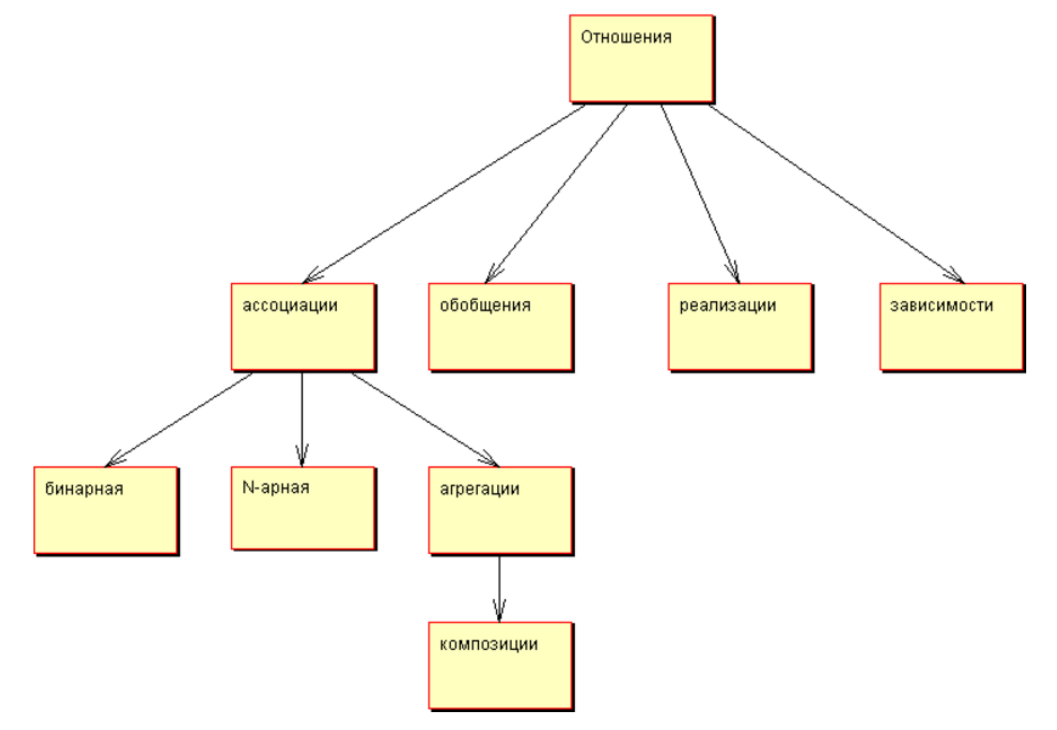
* **Синхронное сообщение**
  + актор-отправитель передаёт ход управления актёру-получателю, которому необходимо провести в прецеденте некоторое действие. Пока проводимое актором-получателем действие не будет завершено (соответственно, не будет получено ответное сообщение), актёр-отправитель теряет возможность производить какие-либо действия.
* **Ответное сообщение**
  + данное сообщение является ответом на синхронное сообщение. Обычно, содержит какое-либо возвращаемое изначальному актору-отправителю значение, также возвращающее ему управление (возможность действовать).
* **Асинхронное сообщение**
  + актор-отправитель передаёт ход управления актору-получателю, которому необходимо провести в прецеденте некоторое действие. Основное отличие от синхронного сообщения состоит в том, что актор-отправитель не теряет возможность совершать другие действия.

**

**Стереотипы сообщений**

* **"call" (вызвать)**
* **"return" (возвратить)**
* **"create" (создать)**
* **"destroy" (уничтожить)**
* **"send" (послать)**

# 24. Отношения UML (виды связей)



Существует четыре типа связей в UML:

* **зависимость**
* **реализация**
* **обобщение**
* **ассоциация**

***1) Зависимость*** – связь между двумя элементами модели, в которой изменение одного элемента может привести к изменению другого элемента . Графически представлена пунктирной линией, иногда со стрелкой, направленной к той сущности, от которой зависит.

Зависимость

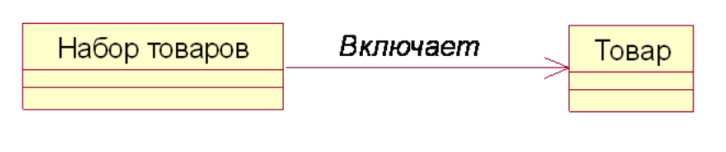
Зависимость – это связь ***использования***, указывающая, что изменение спецификаций одной сущности может повлиять на другие сущности, которые используют ее.

***2) Реализация*** – между классами, когда один из них (**поставщик**) определяет соглашение, которого второй (**клиент**) обязан придерживаться. Это связи между интерфейсами и классами, которые реализуют эти интерфейсы. Это, своего рода, отношение «целое-часть». Поставщик, как правило, представлен абстрактным классом. В графическом исполнении связь реализации – это *гибрид связей обобщения и зависимости*: треугольник указывает на поставщика, а второй конец пунктирной линии – на клиента.

Реализация

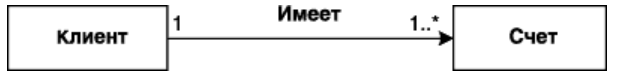
***3) Обобщение*** – выражает ***наследование***, в котором специализированный элемент (потомок) строится по спецификациям обобщенного элемента (родителя). *Потомок разделяет структуру и поведение родителя*. Графически обобщение представлено в виде сплошной линии с пустой стрелкой, указывающей на родителя.

Обобщение

***4)*** С помощью ***ассоциации*** определяется взаимосвязь объектов одной сущности (класса) с объектами другой сущности. Наиболее часто для связывания двух классов используются бинарные ассоциации..

Ассоциация подразделяется на ***бинарную***, ***N-нарную*** и ***агрегацию***

В качестве простого примера направленной ***бинарной*** ассоциации можно рассмотреть отношение между двумя классами - классом Клиент и классом Счет ([рис. 6.2](https://intuit.ru/studies/courses/32/32/lecture/1010?page=1#image.6.2)). Они связаны между собой бинарной ассоциацией с именем Имеет, для которой определен порядок следования классов. Это означает, что конкретный объект класса Клиент всегда должен указываться первым при рассмотрении взаимосвязи с объектом класса Счет.

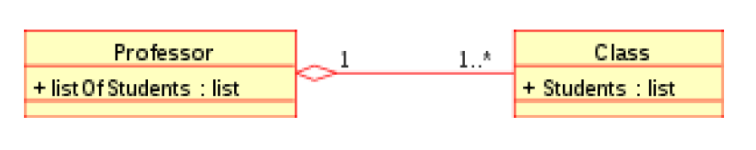


***n-арная*** ассоциация- ассоциация между тремя и большим числом классов

Такая ассоциация связывает отношением более чем три класса, при этом класс может участвовать в ассоциации более чем один раз. Каждый экземпляр n-арной ассоциации представляет собой n-арный кортеж, состоящий из объектов соответствующих классов.



***Агрегацией*** является разновидность ассоциации, которая определяет отношение между целым и его частями. Агрегация не может определять отношение более двух классов (**контейнер и содержимое** (дом и квартиры))



Графически агрегация представляется пустым ромбом на блоке класса «целое», и линией, идущей от этого ромба к классу «часть».

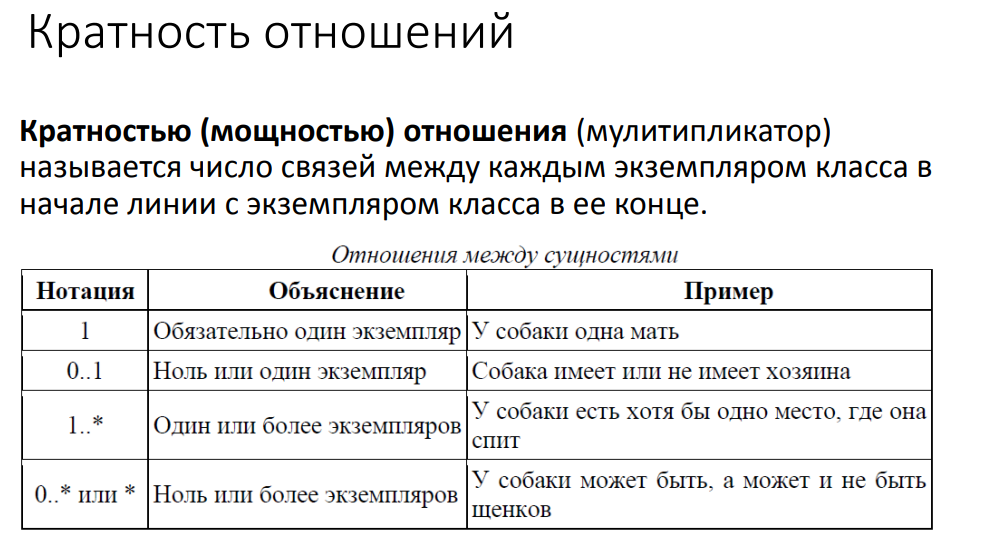
Агрегация

***Композиция*** — более строгий вариант агрегации. Известна также как агрегация по значению.

Композиция – это форма агрегации с четко выраженными отношениями владения и совпадением времени жизни частей и целого. Композиция имеет жесткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.

Графически представляется как и агрегация, но с закрашенным ромбиком.

Композиция



# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 25. Шаблон MVC

MVC как model-view-controller (модель-представление-контроллер).

Это ***способ организации кода***, который предполагает ***выделение блоков***, отвечающих за решение разных задач. Один блок отвечает за данные приложения, другой отвечает за внешний вид, а третий контролирует работу приложения. Компоненты MVC:

* Модель — этот компонент отвечает за данные, а также определяет структуру приложения. Например, если вы создаете To-Do приложение, код компонента model будет определять список задач и отдельные задачи.
* Представление — этот компонент отвечает за взаимодействие с пользователем. То есть код компонента view определяет внешний вид приложения и способы его использования.
* Контроллер — этот компонент отвечает за связь между model и view. Код компонента controller определяет, как сайт реагирует на действия пользователя. По сути, это мозг MVC-приложения.

# 26. Что такое архитектура ПО? (понимание)

Архитектура программного обеспечения- совокупность важнейших решений об организации программной системы. Архитектура включает:

-выбор структурных элементов и их интерфейсов, с помощью которых составлена система, а также их поведения в рамках сотрудничества структурных элементов;

-соединение выбранных элементов структуры и поведения во всё более крупные системы;

-архитектурный стиль, который направляет всю организацию — все элементы, их интерфейсы, их сотрудничество и их соединение.

# 27. Язык UML (назначение)

Язык UML — это графический язык моделирования общего назначения, предназначенный для спецификации, визуализации, проектирования и документирования всех артефактов, создаваемых при разработке программных систем.

Язык UML предназначен для решения следующих задач:

1. Предоставить в распоряжение пользователей легко воспринимаемый и выразительный язык визуального моделирования, специально предназначенный для разработки и документирования моделей сложных систем самого различного целевого назначения.

2. Снабдить исходные понятия языка UML возможностью расширения и специализации для более точного представления моделей систем в конкретной предметной области.

3. Описание языка UML должно поддерживать такую спецификацию моделей, которая не зависит от конкретных языков программирования и инструментальных средств проектирования программных систем.

4. Описание языка UML должно включать в себя семантический базис для понимания общих особенностей ООАП.

5. Поощрять развитие рынка объектных инструментальных средств.

6. Способствовать распространению объектных технологий и соответствующих понятий ООАП.

7. Интегрировать в себя новейшие и наилучшие достижения практики ООАП.

# 28. Класс, объект, атрибут, метод

**Класс** — это элемент ПО, описывающий абстрактный тип данных и его частичную или полную реализацию. Если проще: класс - шаблон кода, по которому создается объект.

**Объект (**экземпляр класса) - это сущность, способная сохранять свое состояние (информацию) и обеспечивающая набор операций (поведение) для проверки и изменения этого состояния.

**Атрибут -** это значение, характеризующее объект в его классе. Среди атрибутов различаются *постоянные* ***атрибуты*** (константы) и ***переменные атрибуты*.** Постоянные атрибуты характеризуют объект в его классе (например, номер счета, категория, имя человека и т.п.). Текущие значения переменных атрибутов характеризуют текущее *состояние* объекта (например, баланс счета, возраст человека и т.п.); изменяя значения этих атрибутов, мы изменяем состояние объекта.

**Метод** - это [функция](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) или [процедура](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), принадлежащая какому-то [классу](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) или [объекту](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).Различают простые методы и статические методы (методы класса):

* простые методы имеют доступ к данным объекта (конкретного экземпляра данного класса),
* статические методы не имеют доступа к данным объекта, и для их использования не нужно создавать экземпляры (данного класса).

Методы предоставляют интерфейс, при помощи которого осуществляется доступ к данным объекта некоторого класса, тем самым, обеспечивая [инкапсуляцию данных](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

В зависимости от того, какой уровень доступа предоставляет тот или иной метод, выделяют:

* открытый (public) интерфейс — общий интерфейс для всех пользователей данного класса;
* защищённый (protected) интерфейс — внутренний интерфейс для всех наследников данного класса;
* закрытый (private) интерфейс — интерфейс, доступный только изнутри данного класса.

Такое разделение интерфейсов позволяет сохранять неизменным открытый интерфейс, но изменять внутреннюю реализацию.

# 29. Интерфейс, абстракция, наследование, инкапсуляция, полиморфизм

**Интерфейс** — это совокупность методов и правил взаимодействия элементов системы.Интерфейсы позволяют наладить [множественное наследование](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BD%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) объектов и в то же время решить проблему [ромбовидного наследования](https://ru.wikipedia.org/wiki/%D0%A0%D0%BE%D0%BC%D0%B1%D0%BE%D0%B2%D0%B8%D0%B4%D0%BD%D0%BE%D0%B5_%D0%BD%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)(ситуация в [объектно-ориентированных](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) [языках программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) с поддержкой [множественного наследования](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BD%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

Интерфейсы в [UML](https://ru.wikipedia.org/wiki/UML) используются для визуализации, специфицирования, конструирования и документирования стыковочных [UML-узлов](https://ru.wikipedia.org/wiki/%D0%A3%D0%B7%D0%B5%D0%BB_(UML)) между составными частями системы.

**Абстра́кция** — в объектно-ориентированном программировании это придание объекту характеристик, которые отличают его от всех объектов, четко определяя его концептуальные границы. Основная идея состоит в том, чтобы отделить способ использования составных объектов данных от деталей их реализации в виде более простых объектов

**Инкапсуляция** — свойство программирования, позволяющее пользователю не задумываться о сложности реализации используемого программного компонента (что у него внутри?), а взаимодействовать с ним посредством предоставляемого интерфейса (публичных методов и членов), а также объединить и защитить жизненно важные для компонента данные. При этом пользователю предоставляется только спецификация (интерфейс) объекта. =Сокрытие реализации

**Наследование** — один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с инкапсуляцией, полиморфизмом и абстракцией), позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом.

**Полиморфи́зм** — возможность объектов с одинаковой спецификацией иметь различную реализацию.Полиморфизм позволяет писать более абстрактные программы и повысить коэффициент повторного использования кода. Общие свойства объектов объединяются в систему, которую могут называть по-разному — интерфейс, класс.

Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций».

# 30. Диаграмма компонентов и диаграмма развертывания

Диагра́мма компоне́нтов, Component diagram — статическая

структурная диаграмма, показывает разбиение программной

системы на структурные компоненты и связи (зависимости)

между компонентами. В качестве физических компонентов могут

выступать файлы, библиотеки, модули, исполняемые файлы,

пакеты и т. п.

##### 

Диагра́мма развёртывания моделирует физическое развертывание артефактов на узлах.

Артефакты - в широком смысле это некие объекты создаваемые в ходе разработки ПО

**Два типа узлов**

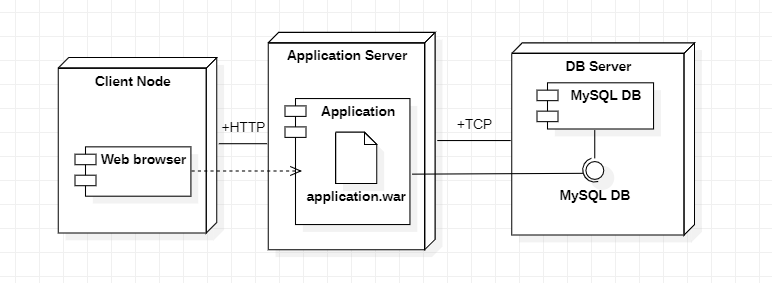
• Узлы устройств — это физические вычислительные ресурсы со

своей памятью и сервисами для выполнения программного

обеспечения, такие как обычные ПК

• Узел среды выполнения — это программный вычислительный

ресурс, который работает внутри внешнего узла и который

предоставляет собой сервис, выполняющий другие исполняемые

программные элементы.