ПРОГРАММНАЯ ИНЖЕНЕРИЯ

АНАЛИЗ И ПРОЕКТИРОВАНИЕ ПО

ПРОЕКТИРОВАНИЕ ПО

ЧТО ТАКОЕ ПРОЕКТИРОВАНИЕ И АРХИТЕКТУРА?

- **Проектирование ПО** – это осознанный выбор решений о логической организации составных частей программного комплекса.
- Проект может быть представлен в виде модели UML, неформального документа или набросков представляющих определенные аспекты дизайна.
- Проектирование – это творческий процесс, который тяжело описать в виде формальных структурированных методов.
- **Архитектура** — это **организация системы**, воплощенная в ее **компонентах**, их **отношениях** между собой и с **окружением**, а также определяет принципы ее проектирования и развития. (IEEE 1471)

ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ

Разработка и документация архитектуры ПО обеспечивает следующие преимущества:

1. **Взаимодействие с заказчиком**
2. **Системный анализ**
3. **Высокоуровневое повторное использование ПО**

ВЛИЯНИЕ НЕФУНКЦИОНАЛЬНЫХ ТРЕБОВАНИЙ НА АРХИТЕКТУРУ ПО

- **Производительность и эффективность** – локализация критически-важных операций
- **Безопасность** – слоистая архитектура
- **Высокая доступность** – избыточность используемых компонентов
- **Расширяемость и гибкость** – мелкомодульная архитектура слабосвязанных компонентов

Паттерны архитектуры ПО

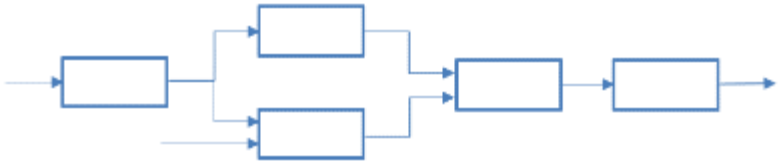
АРХИТЕКТУРНЫЕ ПАТТЕРНЫ

- Не существует формального перечня существующих архитектурных паттернов
- Но на сегодняшний день можно выделить основные (часто-используемые) подходы:
  - Каналы и фильтры
  - Многоуровневая архитектура
  - Архитектура, управляемая событиями (Event-Driven Architecture)
  - Микроядерная архитектура
  - Микросервисная архитектура

Немного об архитектурах программного обеспечения
https://habrahabr.ru/company/it-grad/blog/276297/

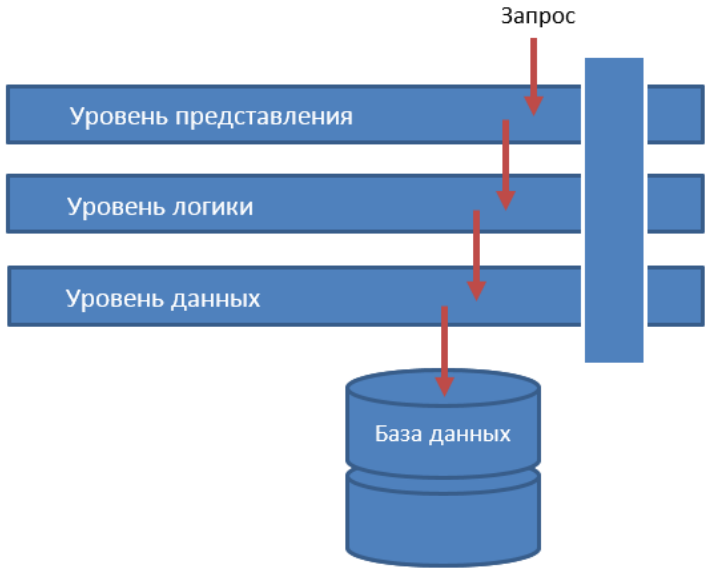
Каналы и фильтры

- Подходит в том случае, если процесс работы приложения распадается на несколько шагов, которые могут выполнятся отдельными обработчиками.
- Основными компонентами являются «фильтр» (filter) и «канал» (pipe). Иногда дополнительно выделяют «источник данных» (data source) и «потребитель данных» (data sink).
  - Каналы обеспечивают передачу данных и синхронизацию.
  - Фильтр же принимает на вход данные и обрабатывает их, трансформируя в некое иное представление, а затем передает дальше (например, шифр Цезаря)
- В качестве примера использования этой архитектуры может служить оболочка UNIX Shell



Многоуровневая (слоистая) архитектура

- Является одной из самых известных архитектур, в которой каждый слой выполняет определенную функцию. В зависимости от ваших нужд вы можете реализовать любое количество уровней.



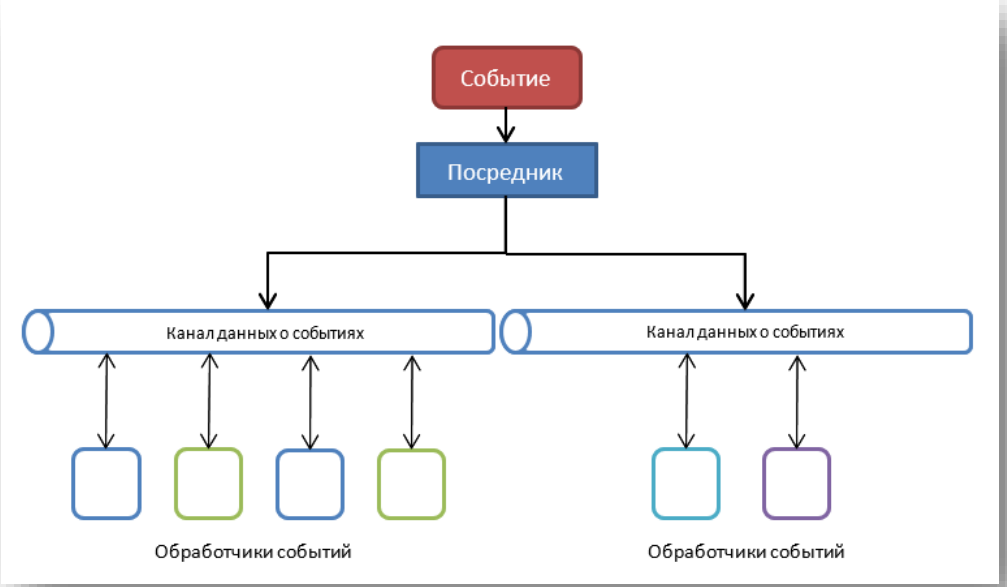
- Является одной из самых известных архитектур, в которой каждый слой выполняет определенную функцию. В зависимости от ваших нужд вы можете реализовать любое количество уровней.

- Достоинствами применения такой архитектуры являются простота разработки (в основном из-за того, что этот вид архитектуры всем знаком) и простота тестирования.
- Среди недостатков можно выделить возможные сложности с производительностью и масштабированием – всему виной необходимость прохождения запросов и данных по всем уровням (опять же, в том случае, если все слои являются закрытыми).

Немного об архитектурах программного обеспечения https://habrahabr.ru/company/it-grad/blog/276297/

Архитектура, управляемая событиями (Event-Driven Architecture)

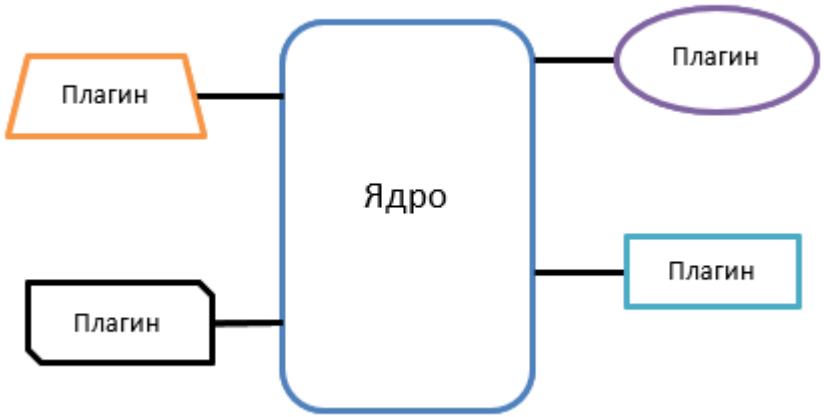
- Широко используется для создания масштабируемых систем.
- Обработчики являются изолированными независимыми компонентами, отвечающими (в идеале) за какую-нибудь одну задачу, и содержат бизнес-логику, необходимую для работы.



- Архитектура, управляемая событиями – это относительно сложный паттерн. Причиной тому – его **распределенная и асинхронная природа**. Вам придется решать проблемы фрагментации сети, обрабатывать ошибки очереди событий и так далее.
- Плюсами этой архитектуры могут служить **высокая производительность**, **легкость развертки** и **поразительные возможности масштабирования**. Однако возможно усложнение процесса тестирования системы.

Микроядерная архитектура

- Паттерн состоит из двух компонентов: основной системы (ядра) и плагинов.
- Ядро содержит **минимум бизнес-логики**, но руководит **загрузкой**, **выгрузкой** и **запуском** необходимых плагинов. Таким образом, плагины оказываются несвязанными друг с другом.



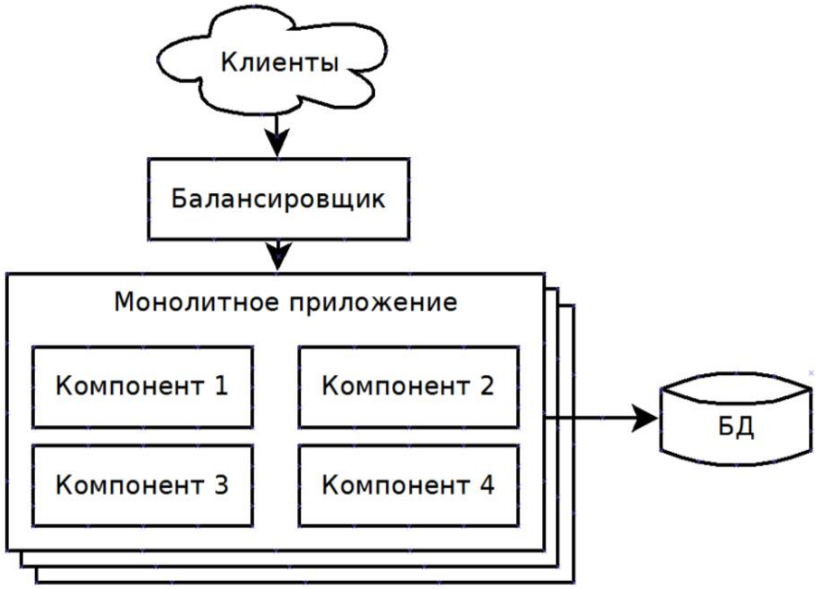
- Поскольку плагины могут разрабатываться независимо друг от друга, такие системы обладают очень высокой гибкостью и, как следствие, легко тестируются. Производительность приложения, построенного на основе такой архитектуры, напрямую зависит от количества подключенных и активных модулей.
- Возможно самым лучшим примером микроядерной архитектуры будет Eclipse IDE. Скачивая Eclipse без надстроек, вы получаете совершенно пустой редактор.

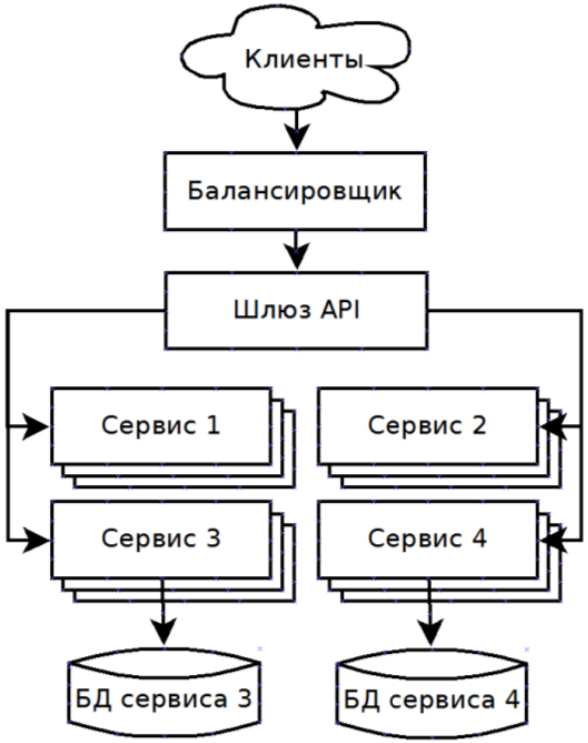
Немного об архитектурах программного обеспечения https://habrahabr.ru/company/it-grad/blog/276297/

Микросервисная архитектура

- **Микросервисная архитектура** – это паттерн проектирования облачных приложений, подразумевающий, что сложное приложение разделяется на ряд небольших независимых сервисов, взаимодействующих друг с другом посредством кроссплатформенного API.
- Каждый микросервис включает в себя бизнес-логику и представляет собой совершенно независимый компонент. Сервисы одной системы могут быть написаны на различных языках программирования и общаться друг с другом, используя различные протоколы.
- *Отличительные особенности микросервисов:*
  - Микросервисы независимы;
  - Микросервисы общаются друг с другой только при помощи сообщений;
  - Каждый микросервис может быть развернут, приостановлен, дублирован или перемещен независимо от других.

Микросервисная архитектура VS Монолитная архитектура

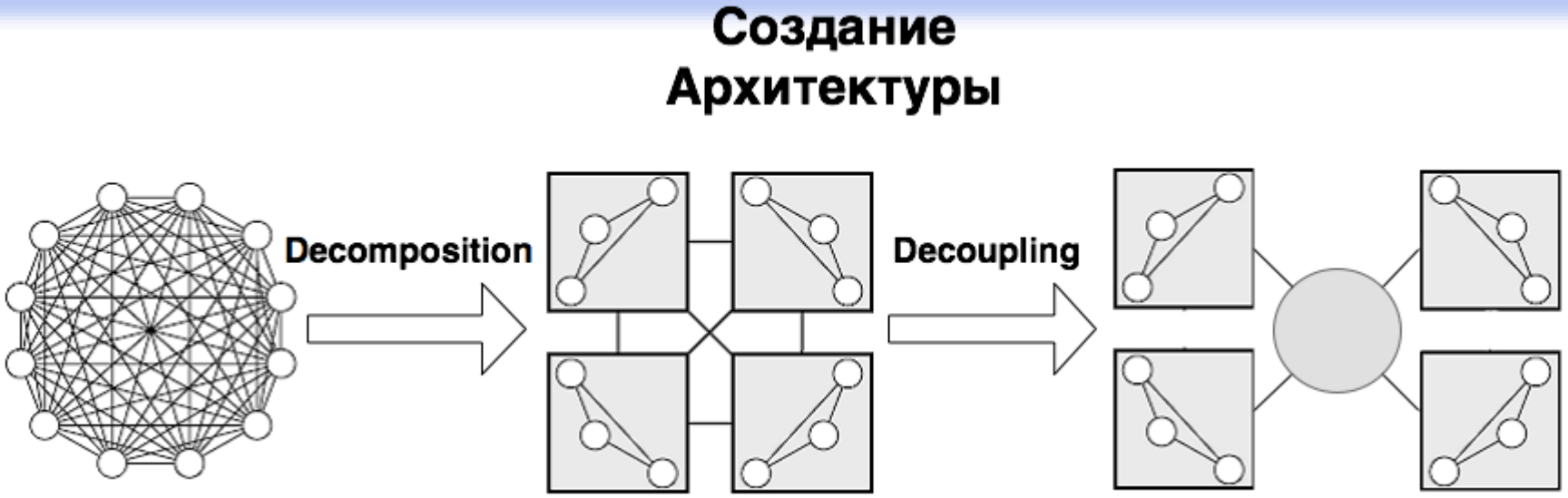




- Достоинства
  - сверхвысокая масштабируемость
  - легкость распределения задач
- Недостатки
  - необходимость передачи большого объема данных
  - необходимость автоматизации развертывания и тестирования

Модульная архитектура

Создание модульной архитектуры

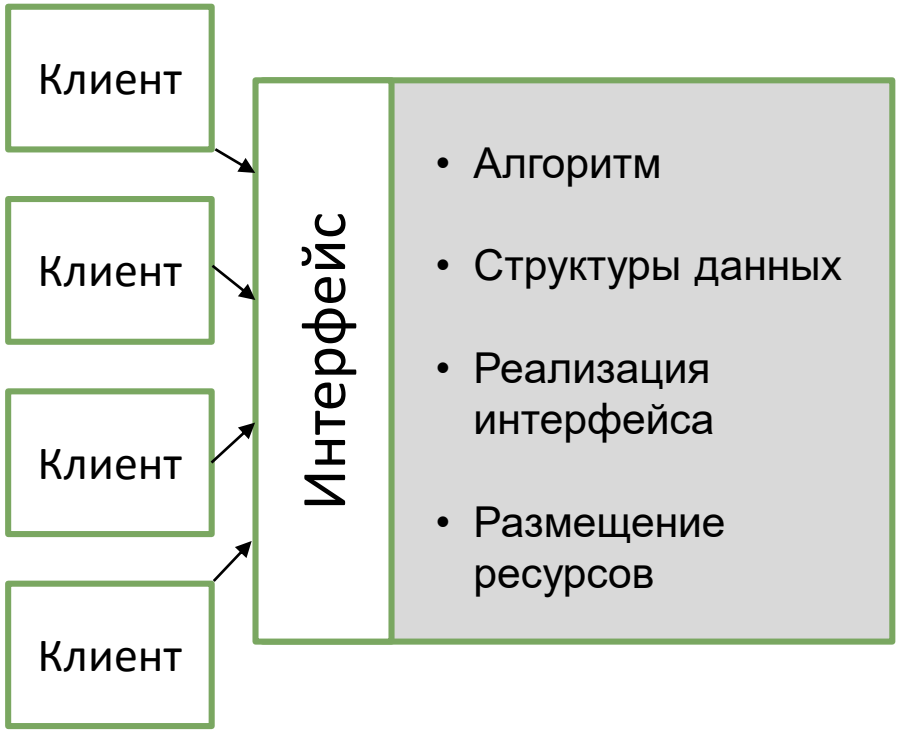


- Главная задача при разработке больших систем - снижение сложности.
- Принцип: **«разделяй и властвуй» (divide et impera)**, но по сути речь идет об иерархической декомпозиции.
- Помимо снижения сложности, обеспечивается гибкость системы, возможности для масштабирования, повышается устойчивость за счет дублирования критически важных частей.

Создание архитектуры программы или как проектировать табуретку
https://habrahabr.ru/post/276593/

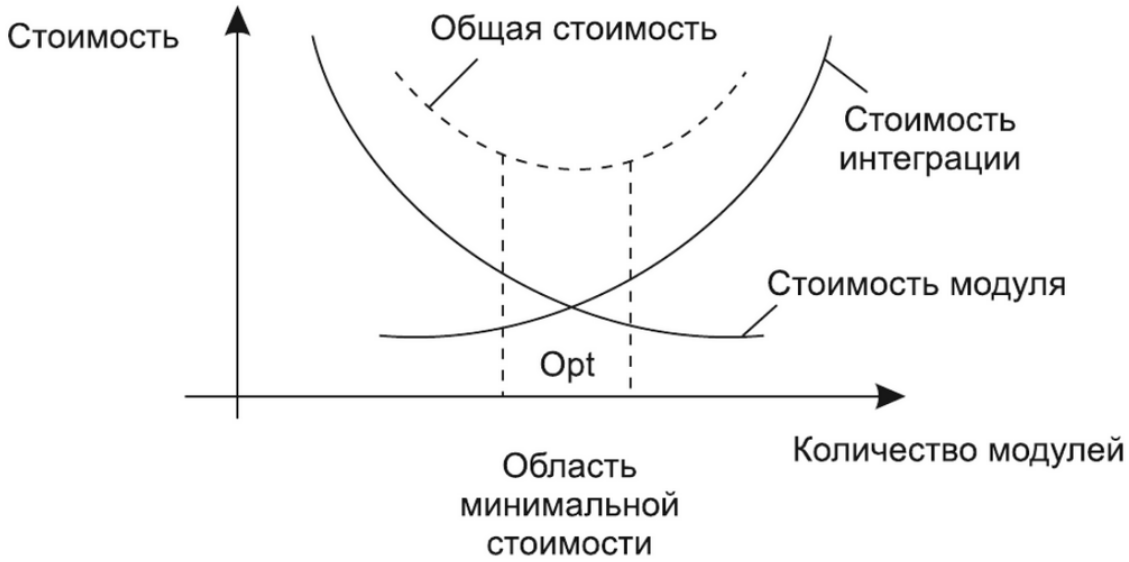
Модульность

- Программная система делится *на именуемые и адресуемые компоненты*, часто называемые *модулями*, которые затем интегрируются для совместного решения проблемы.
- **Модуль** - фрагмент программного текста, являющийся строительным блоком для физической структуры системы.
- Как правило, модуль состоит из интерфейсной части и части-реализации.



Модульность

- Модульность — свойство ПО, обеспечивающее интеллектуальную возможность создания сколь угодно сложной программы.
- Но необходимо учитывать затраты на дальнейшую интеграцию модулей.



Информационная закрытость

- Информационная закрытость означает, что:
  - Все модули независимы и обмениваются информацией только необходимой для работы
  - Доступ к операциям и структурам данных модуля ограничен.
- Это позволяет:
  - Обеспечить разработку модулей различными независимыми коллективами;
  - Обеспечить легкую модификацию системы
- Идеальный модуль – это черный ящик, содержимое которого не видно клиенту.

Достоинства хорошей архитектуры

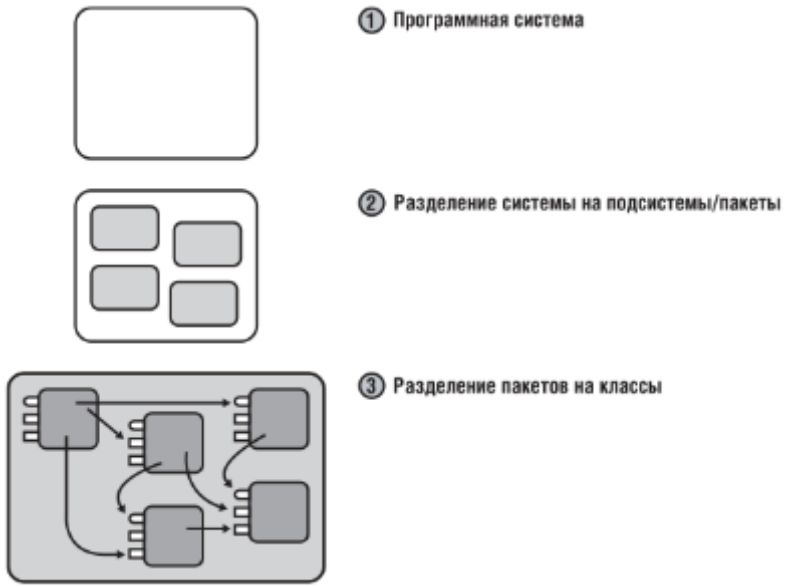
- **Масштабируемость (Scalability)**
  возможность расширять систему и увеличивать ее производительность, за счет добавления новых модулей.
- **Ремонтопригодность (Maintainability)**
  изменение одного модуля не требует изменения других модулей
- **Заменимость модулей (Swappability)**
  модуль легко заменить на другой
- **Возможность тестирования (Unit Testing)**
  модуль можно отсоединить от всех остальных и протестировать / починить
- **Переиспользование (Reusability)**
  модуль может быть переиспользован в других программах и другом окружении
- **Сопровождаемость (Maintenance)**
  разбитую на модули программу легче понимать и сопровождать

Создание архитектуры программы или как проектировать табуретку
https://habrahabr.ru/post/276593/

ПОДХОДЫ К МОДУЛЬНОЙ ДЕКОМПОЗИЦИИ

ДЕКОМПОЗИЦИЯ: ИЕРАРХИЧЕСКАЯ

- Декомпозиция должна быть иерархической:
  - сначала систему разбивают на крупные функциональные модули/подсистемы, описывающие ее работу в самом общем виде.
  - Затем, полученные модули, анализируются более детально и, в свою очередь, делятся на под-модули либо на объекты.
- Типичные модули первого уровня:
  - «бизнес-логика»,
  - «пользовательский интерфейс»
  - «доступ к БД»
  - «связь с конкретным оборудованием или ОС».
- Для обозримости на каждом иерархическом уровне рекомендуют выделять от 2 до 7 модулей.



ДЕКОМПОЗИЦИЯ: ФУНКЦИОНАЛЬНАЯ

- Декомпозиция должна быть функциональной:
  - Каждый модуль должен отвечать за решение какой-то подзадачи и выполнять соответствующую ей **функцию**.
  - Помимо функционального назначения модуль характеризуется также набором данных, необходимых ему для выполнения его функции.
- Модуль — это не произвольный кусок кода, а отдельная функционально осмысленная и законченная программная единица(подпрограмма), которая обеспечивает решение некоторой задачи и в идеале может работать самостоятельно или в другом окружении и быть переиспользуемой. Модуль должен быть некой "целостностью, способной к относительной самостоятельности в поведении и развитии"

High Cohesion + Low Coupling

- При декомпозиции необходимо соблюдать
  - **максимальную внутреннюю** связность (Cohesion) модулей: модуль должен быть сфокусирован на решении одной узкой проблемы, а не занимается выполнением разнородных функций или несвязанных между собой обязанностей.
  - **минимальную внешнюю** связанность (Coupling) модулей: модули, на которые разбивается система, должны быть, по возможности, независимы или *слабо связанны* друг с другом. Они должны иметь возможность взаимодействовать, но при этом *как можно меньше знать друг о друге* (**принцип минимального знания**).
- Это позволит спроектировать каждый модуль как «черный ящик», т.е. каждый модуль можно будет разрабатывать или модифицировать независимо от остальных компонентов системы.

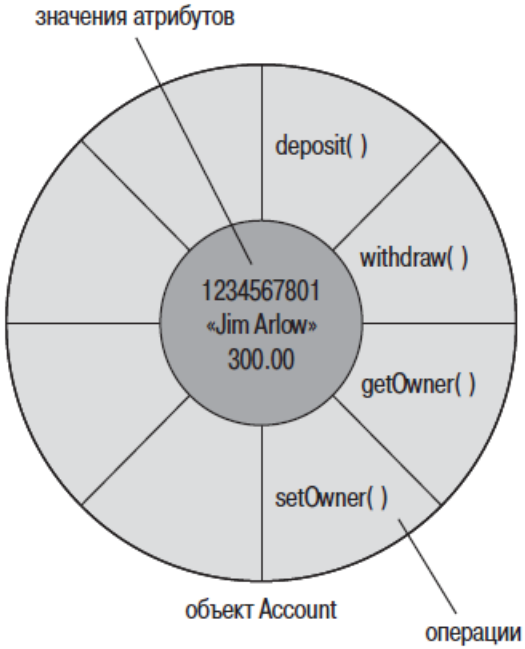
Признаки хорошо-спроектированных модулей

- **функциональная целостность и завершенность** — каждый модуль реализует одну функцию, но реализует хорошо и полностью;
- **один вход и один выход** — на входе программный модуль получает определенный набор исходных данных, выполняет содержательную обработку и возвращает один набор результатных данных
- **логическая независимость** — результат работы программного модуля зависит только от исходных данных, но не зависит от работы других модулей;
- **слабые информационные связи с другими модулями** — обмен информацией между модулями должен быть по возможности минимизирован.

ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ

ОО-ПРОЕКТИРВОАНИЕ

- ОО-проектирование основывается на концепции объекта: сущности, сохраняющей свое внутреннее состояние и поддерживающей операции для управления внутренним состоянием.



ПРОЦЕСС ОБЪЕКТНО-ОРИЕНТИРОВАННОГО РЕШЕНИЯ ЗАДАЧИ

- ОО анализ – формирование объектно-ориентированной модели предметной области.
- ОО проектирование – развитие ОО модели программной системы, для достижения определенных требований.
- ОО реализация – реализация разработанной модели посредством ОО языка программирования.

SOLID

- **SOLID** - аббревиатура пяти основных принципов дизайна классов в объектно-ориентированном проектировании
  - *Single responsibility:* На каждый объект должна быть возложена одна единственная обязанность.
  - *Open-closed:* Программные сущности должны быть открыты для расширения, но закрыты для изменения.
  - *Liskov substitution:* Классы в программе должны поддерживать замену их наследниками без изменения свойств программы.
  - *Interface segregation:* Много специализированных интерфейсов лучше, чем один универсальный.
  - *Dependency inversion:* Модули верхнего уровня не должны завесить от модулей нижнего уровня.

ПРИНЦИП ЕДИНСТВЕННОЙ ОБЯЗАННОСТИ (*SINGLE RESPONSIBILITY PRINCIPLE*)

- На каждый класс должна быть возложена одна единственная обязанность.
- Все методы и атрибуты класса должны быть ориентированы на реализацию данной обязанности.

ПРИНЦИП ОТКРЫТОСТИ/ЗАКРЫТОСТИ (*OPEN/CLOSED PRINCIPLE*)

- Программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения

- Два варианта значения принципа:

  1. *принцип Мейера:* разработанная реализация класса требует только исправления ошибок, а новые или изменённые функции требуют создания нового класса (можно применять наследование – **наследование реализации**)

  2. *Полиморфный принцип:* наследование может быть только от абстрактных базовых классов. Спецификации интерфейсов могут быть переиспользованы, но реализации изменяться не должны.

ПРИНЦИП ПОДСТАНОВКИ БАРБАРЫ ЛИСКОВ (*LISKOV SUBSTITUTION PRINCIPLE*)

- Функции, которые используют базовый тип, должны иметь возможность использовать подтипы (наследники) базового типа не зная об этом.
- Классы в программе должны поддерживать замену их наследниками без изменения свойств программы.
- Поведение наследуемых классов не должно противоречить поведению, заданному базовым классом, то есть поведение наследуемых классов должно быть ожидаемым для кода, использующего переменную базового типа.

ПРИМЕР ПРОТИВОРЕЧИЯ ПРИНЦИПУ SLP

- Сформируйте иерархию классов, содержащую понятия «прямоугольник» и «квадрат»

```
class Rectangle {
public:
   virtual void setHeight(int newHeight);
   virtual void setWidth(int newWidth);
   virtual int height() const;       // возвращают текущие значения
   virtual int width() const;
...
};
void makeBigger(Rectangle& r)        // функция увеличивает площадь r
{
   int oldHeight = r.height();
   r.setWidth(r.width() + 10);       // увеличить ширину r на 10
   assert(r.height() == oldHeight); // убедиться, что высота r
}                                    // не изменилась
```

ПРИМЕР ПРОТИВОРЕЧИЯ ПРИНЦИПУ SLP (2)

```
class Square: public Rectangle {…};
Square s;
...
assert(s.width() == s.height()); // должно быть справедливо для
                                 // всех квадратов
makeBigger(s); // из-за наследования, s является
               // Rectangle, поэтому мы можем
               // увеличить его площадь
assert(s.width() == s.height()); // По-прежнему должно быть справедливо
                                 // для всех квадратов
```

- Таким образом, принцип SLP – не наследуй квадрат от прямоугольника!

Принцип разделения интерфейсов (*Interface segregation*)

- Много специализированных интерфейсов лучше, чем один универсальный.
- Клиенты не должны зависеть от методов, которые они не используют.
- Слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе.

Пример нарушения принципа разделения интерфейсов

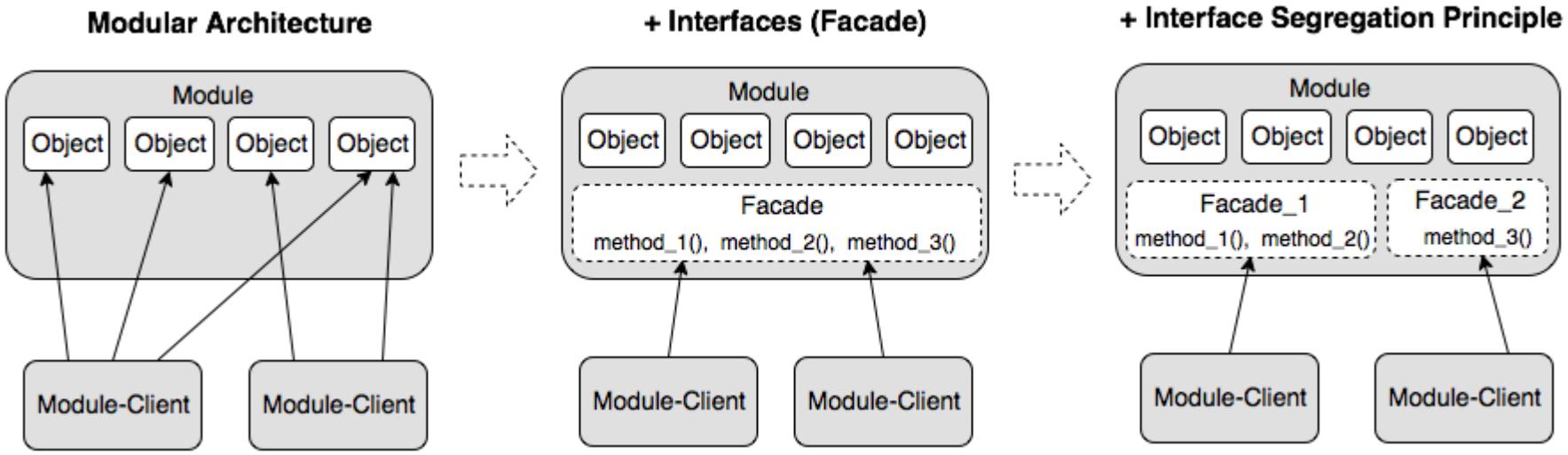
```
public interface Animal {
     void fly();
     void run();
     void bark();
}

public class Bird implements Animal
{
     public void bark() { /* do nothing */ }
     public void run()  {     // write code about running of the bird}
     public void fly()  {     // write code about flying of the bird }
}

public class Cat implements Animal
{
     public void fly()  {throw new Exception("Undefined cat property"); }
     public void bark() {throw new Exception("Undefined cat property"); }
     public void run()  {// write code about running of the cat }
}
```

Реализация интерфейса: фасад

- **Фасад** — это объект-интерфейс, аккумулирующий в себе высокоуровневый набор операций для работы с некоторой подсистемой, скрывающий за собой ее внутреннюю структуру и истинную сложность.
- Обеспечивает защиту от изменений в реализации подсистемы. Служит единой точкой входа — "вы пинаете фасад, а он знает, кого там надо пнуть в этой подсистеме, чтобы получить нужное".

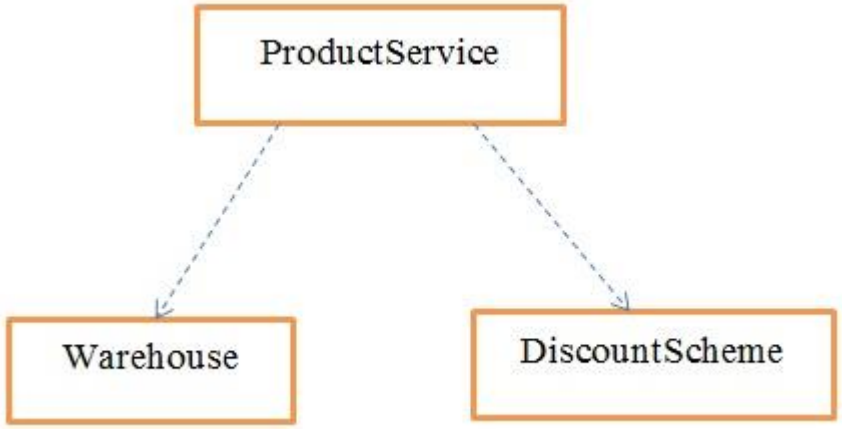


Создание архитектуры программы или как проектировать табуретку
https://habrahabr.ru/post/276593/

ПРИНЦИП ИНВЕРСИИ ЗАВИСИМОСТЕЙ (*DEPENDENCY INVERSION PRINCIPLE*)

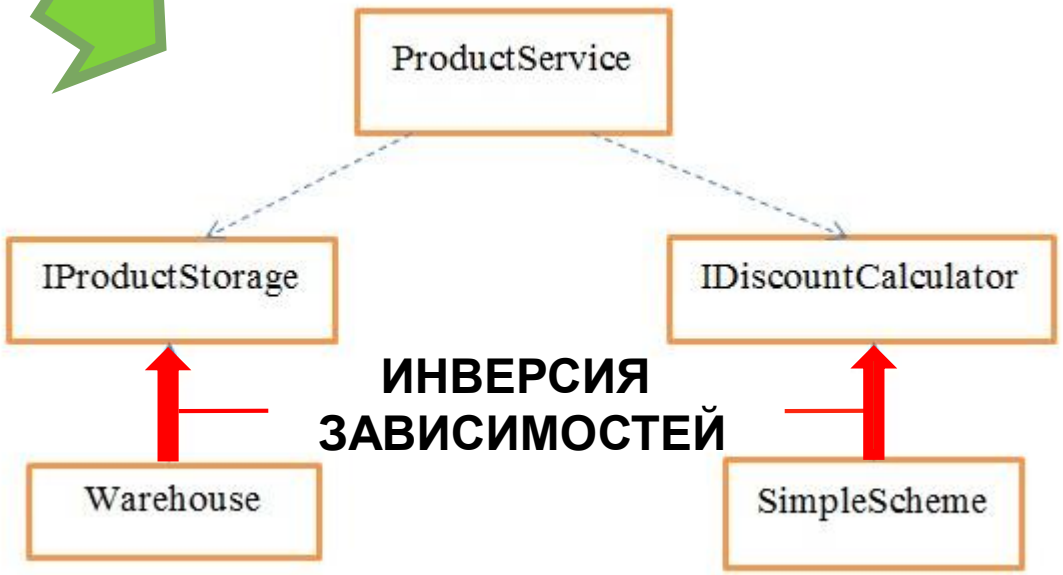
- Принцип обеспечения слабой внешней связанности разрабатываемых модулей:
  - Высокоуровневые модули не должны зависеть напрямую от низкоуровневых модулей. И те, и другие должны зависеть от абстракций
  - Абстракции не должны зависеть от детализации. Детализация должна зависеть от абстракций.
- ***Смысл: все зависимости должны быть в виде интерфейсов***

ПРИМЕР: РАСЧЕТ СКИДОК ДЛЯ ТОВАРОВ НА СКЛАДЕ



Нужно выделить использование реализаций Warehouse и Discount Scheme из ProductService при помощи абстракций.

Мы не можем без изменения ProductService рассчитать скидку на товары, которые могут быть не только на складе. Так же нет возможности подсчитать скидку по другой карте скидок (с другим Disctount Scheme).



ПРОБЛЕМЫ, РЕШАЕМЫЕ ПРИМЕНЕНИЕМ ПРИНЦИПА ИНВЕРСИИ ЗАВИСИМОСТЕЙ

1. **Жесткость ПО:** если изменение одного модуля приводит к изменению других модулей.
2. **Хрупкость ПО:** если изменения в одном модуле приводят к неконтролируемым ошибкам в других частях программы.
3. **Неподвижность ПО:** если модуль сложно отделить от остальной части приложения для повторного использования.

Keep it DRY

- **Don’t Repeat Yourself** *(Не повторяйся)*
  - Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы

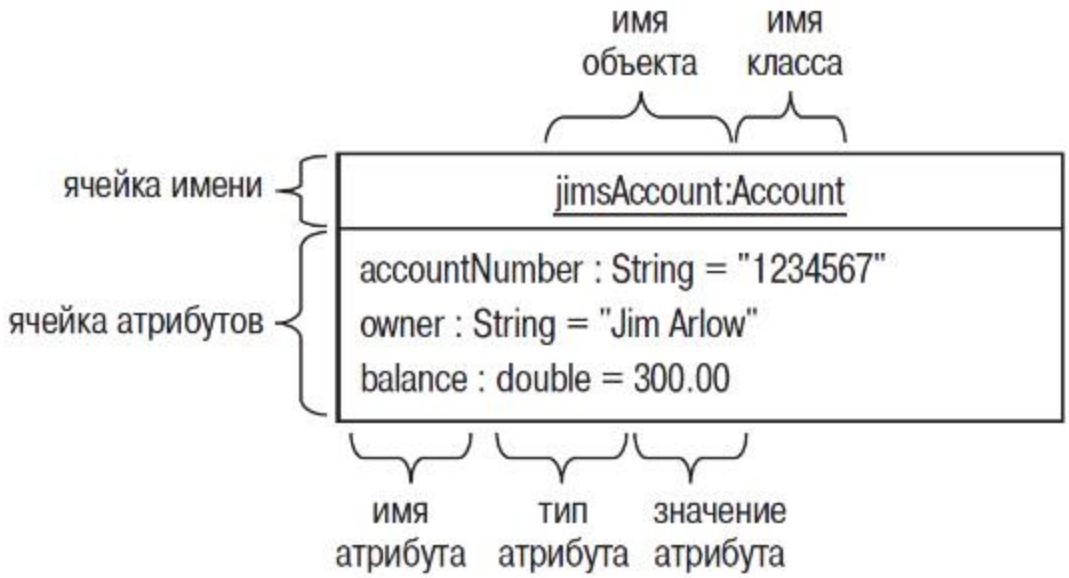
> Энди Хант, Дэйв Томас
> **The Pragmatic Programmer**

  - Применяется к:
    - Исходному коду реализации методов, классов, модулей;
    - Схемам баз данных;
    - Планам тестирования;
    - Документации и т.д.
  - Если DRY применяется успешно, то изменение единственного элемента системы не требует внесения изменений в другие, логически не связанные элементы.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПОСРЕДСТВОМ UML

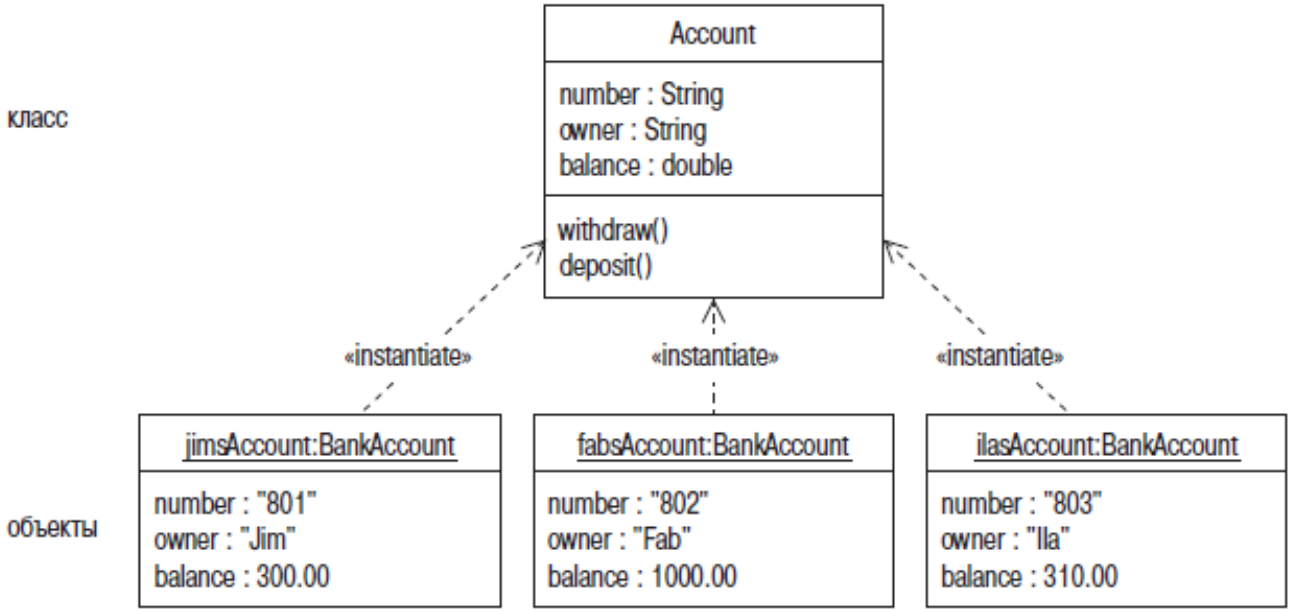
НОТАЦИЯ ОБЪЕКТОВ UML

- Прямоугольник с двумя ячейками:
  - Идентификатор объекта (подчеркнутый) и/или имя класса через двоеточие
  - Блок атрибутов (по выбору, т.к. набор атрибутов определяет класс)



КЛАССЫ И ОБЪЕКТЫ UML

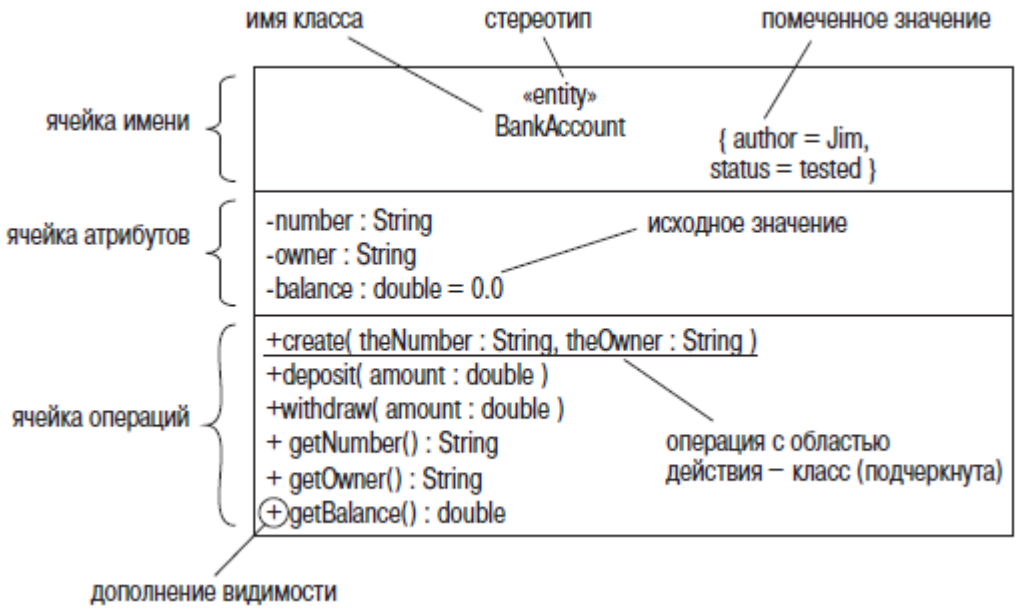
- Между классом и объектами этого класса устанавливается отношение «instantiate» (создать экземпляр)



- Отношение зависимости означает, что изменение сущности поставщика оказывает влияние на сущность клиент

НОТАЦИЯ КЛАССОВ UML

- Обязательной частью в визуальном синтаксисе является только ячейка с именем класса. Все остальные ячейки и дополнения необязательны.



НОТАЦИЯ АТРИБУТОВ КЛАССА

- Единственная обязательная часть UML-синтаксиса для атрибута является его имя.

видимость имя : тип [множественность] = начальноеЗначение

обязателен

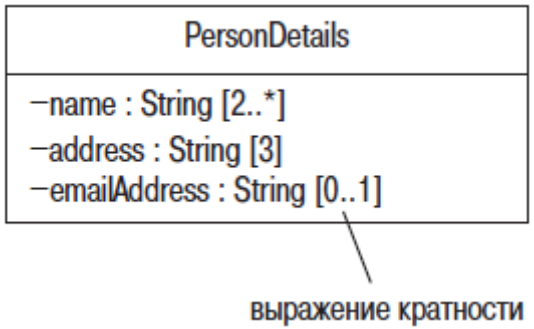
Видимость:

+	Public
-	Private
#	Protected
~	Package

Базовые типы:

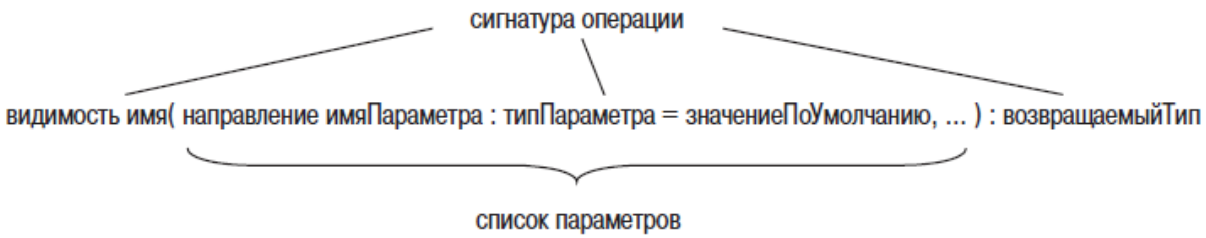
Integer
UnlimitedNatural
Boolean
String
Real

Множественность:



НОТАЦИЯ ОПЕРАЦИЙ

- Сигнатура операции включает имя, тип всех ее параметров и возвращаемый тип



- Направление параметра:
  - in (по умолчанию) – входной параметр
  - inout – параметр ввода/вывода
  - out – выходной параметр
  - return – возвращаемый параметр (UML 2.1 – только 1)

НОТАЦИЯ ОБЛАСТИ ДЕЙСТВИЯ

- Иногда нужно определить атрибуты, которые имеют единственное, общее для *всех объектов класса значение.* И нужны операции (как операции создания объектов), не относящиеся ни к одному конкретному экземпляру класса. Говорят, что такие атрибуты и операции имеют *область действия – класс.*