Focused forward

Netcracker

Lecture 1. Part 1. Introduction to OOP

Alexei Khudnitsky

Знакомство

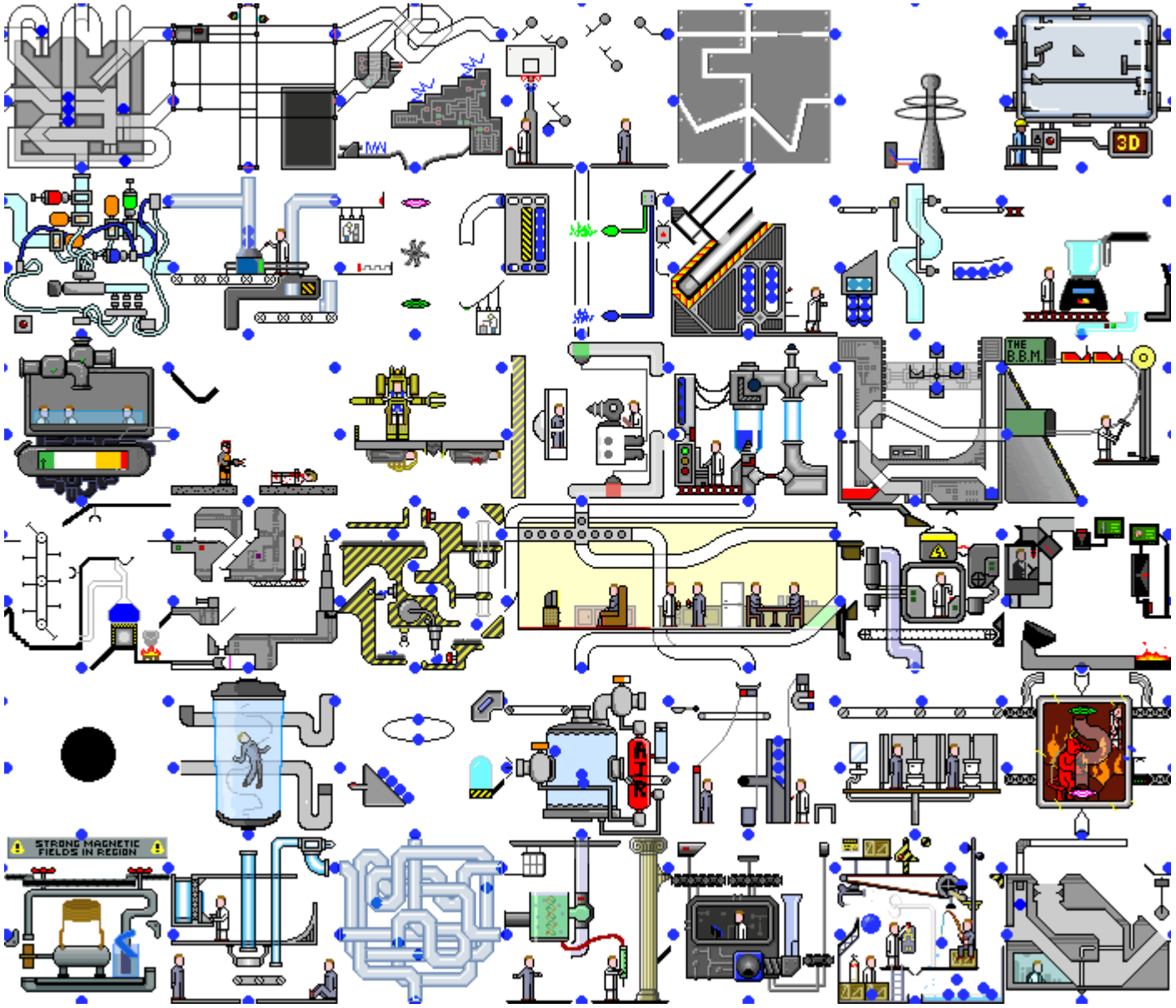
- Худницкий Алексей
- Lead Software Engineer
- Infrastructure Management. Implementation Skill Center.Minsk
- Лектор для групп MANO-1, MANO-2
- Skype: alexei_khudnitsky
- Mail: khudnitsky@gmail.com

Программа

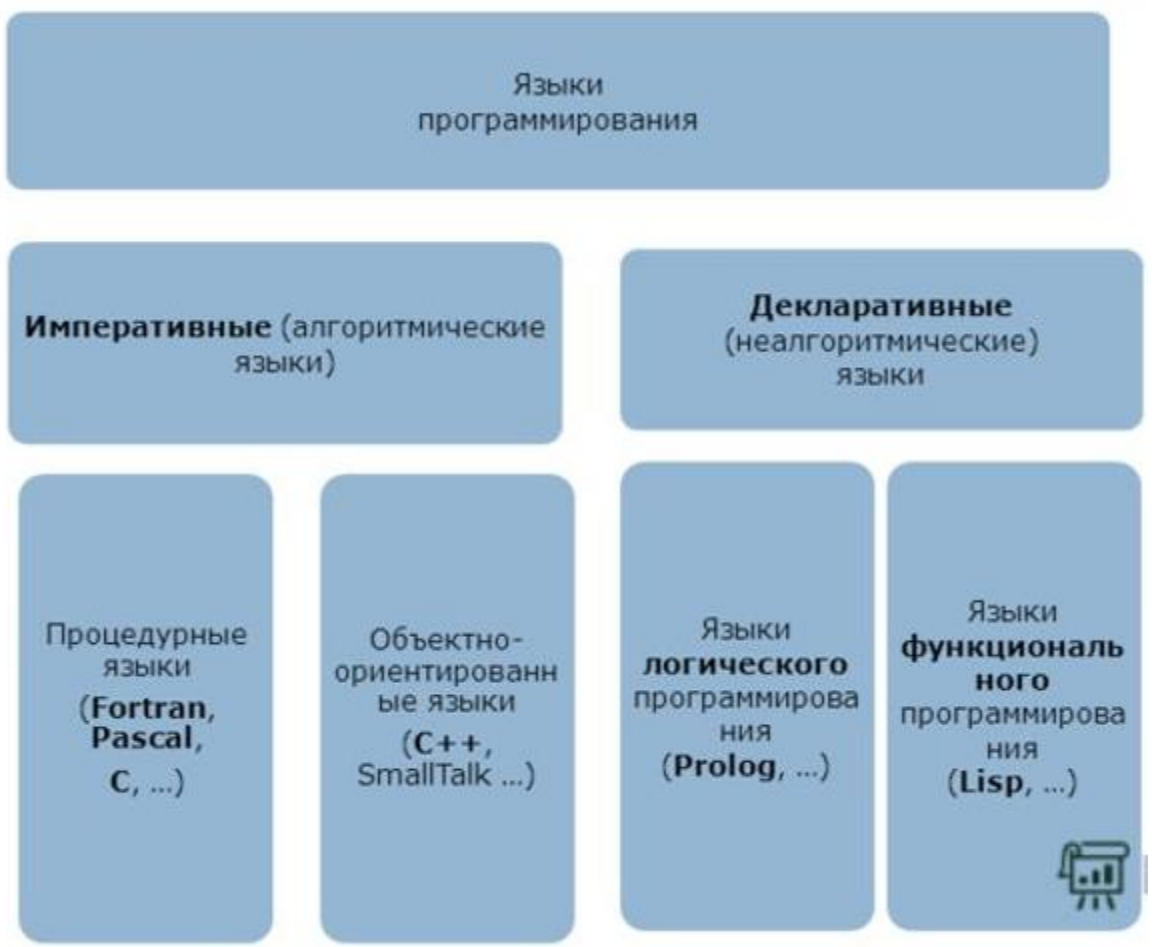
- Introduction to OOP. UML. SOLID Principles
- Application architecture. Introduction to Maven. Introduction to VCS. Java 8 Overview
- Databases Design. Indexes. PKs & FKs. Transactions. Isolation levels
- JDBC. DAO & DTO.
- Refactoring. Design Patterns. TDD. JUnit. Mockito
- Logging. Servlets & JSPs.
- Spring Framework. Spring MVC. Introduction.
- HTTP. JSON. REST. Spring Data JPA. Spring Boot.
- Introduction to microservice architecture. Docker. Openshift. Spring Security.

Программы бывают: простыми и ...

... И сложными



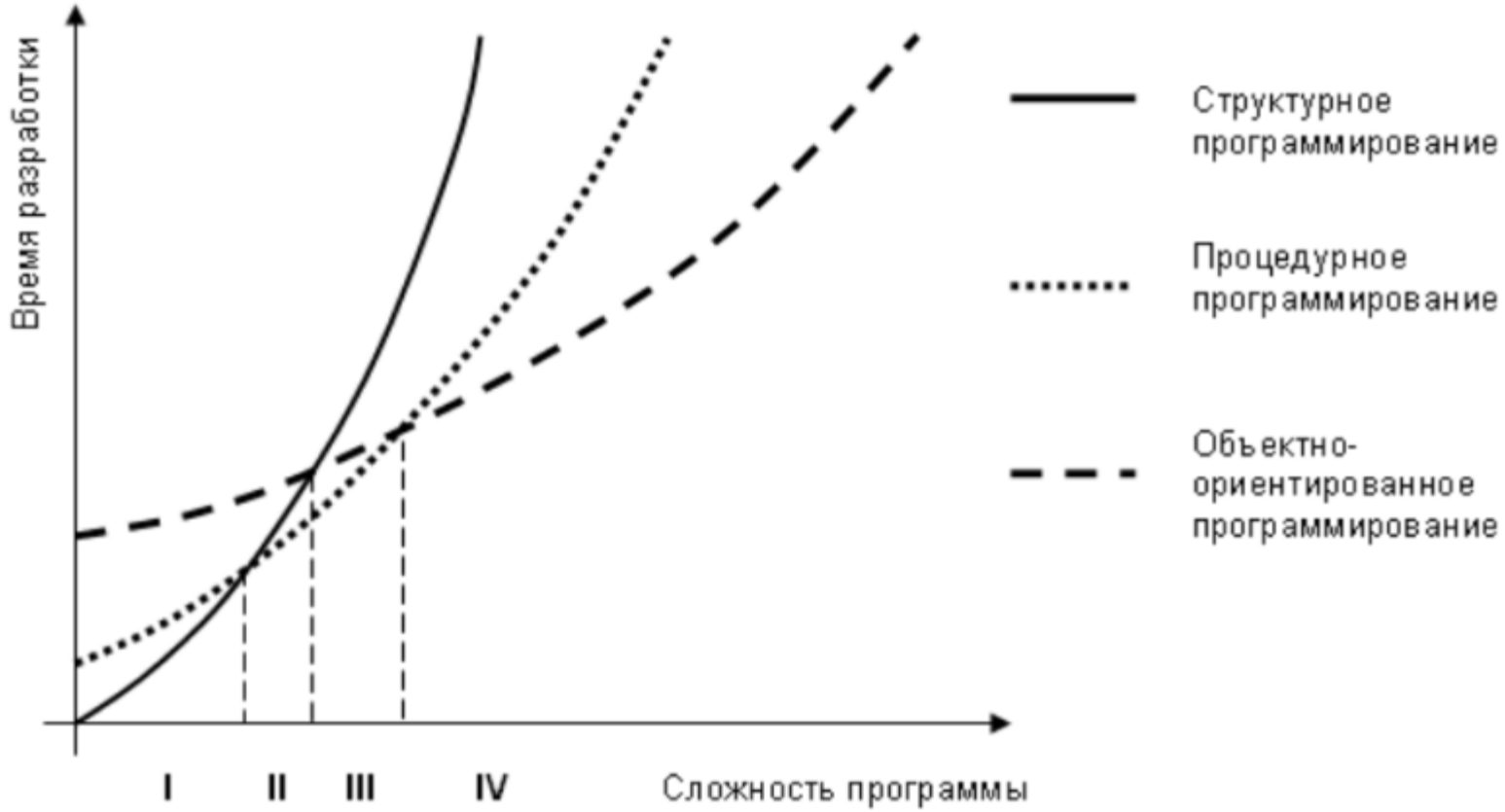
Парадигмы программирования



Парадигмы программирования

Парадигма	Вид абстракции	Задачи
Процедурно-ориентированная	Алгоритмы	Вычислительные
Функционально-ориентированная	Функции	Специфические
Логико-ориентированная	Цели, часто выраженные в терминах исчисления предикатов	Проектирование баз знаний
Объектно-ориентированная	Классы и объекты	Универсальные

Сравнение


Время разработки
I
II
III
IV
Сложность программы
Структурное программирование
Процедурное программирование
Объектно-ориентированное программирование

Проблемы, решаемые ООП

- уменьшение сложности программного обеспечения;
- повышение надежности программного обеспечения;
- обеспечение возможности модификации отдельных компонентов программного обеспечения без изменения остальных его компонентов;
- обеспечение возможности повторного использования отдельных компонентов программного обеспечения.

Ключевые идеи ООП

- Быстрее пишется не та программа, код которой пишется быстро, а та, код которой легче изменяется.

- Программа должна состоять из независимых модулей, которые можно:

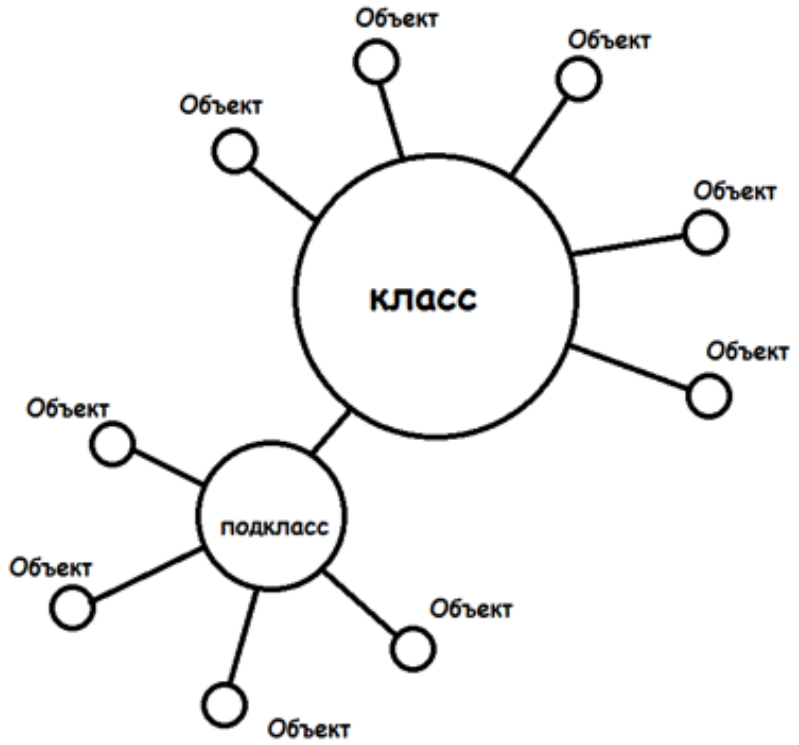
  – разрабатывать и тестировать независимо от остальных частей программы;

  – заменять на более «продвинутые» версии без переделки всей программы;

  – использовать без знаний об их внутреннем представлении

Объектно-ориентированное программирование за 15 секунд.

Значит идея простая до безобразия – тупо писать меньше кода без повторов. Часто используемые куски кода засовываем в одно место под названием Класс и используем его в оставшихся кусках кода под названием Объекты. Ещё можно сделать Подкласс, ну а можно и не делать. Всё.

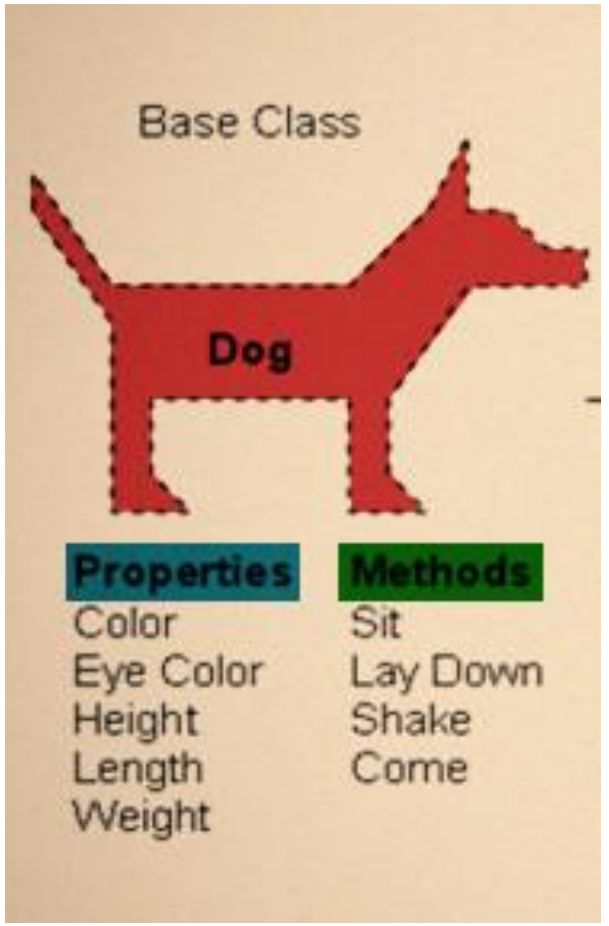


Класс и Объект

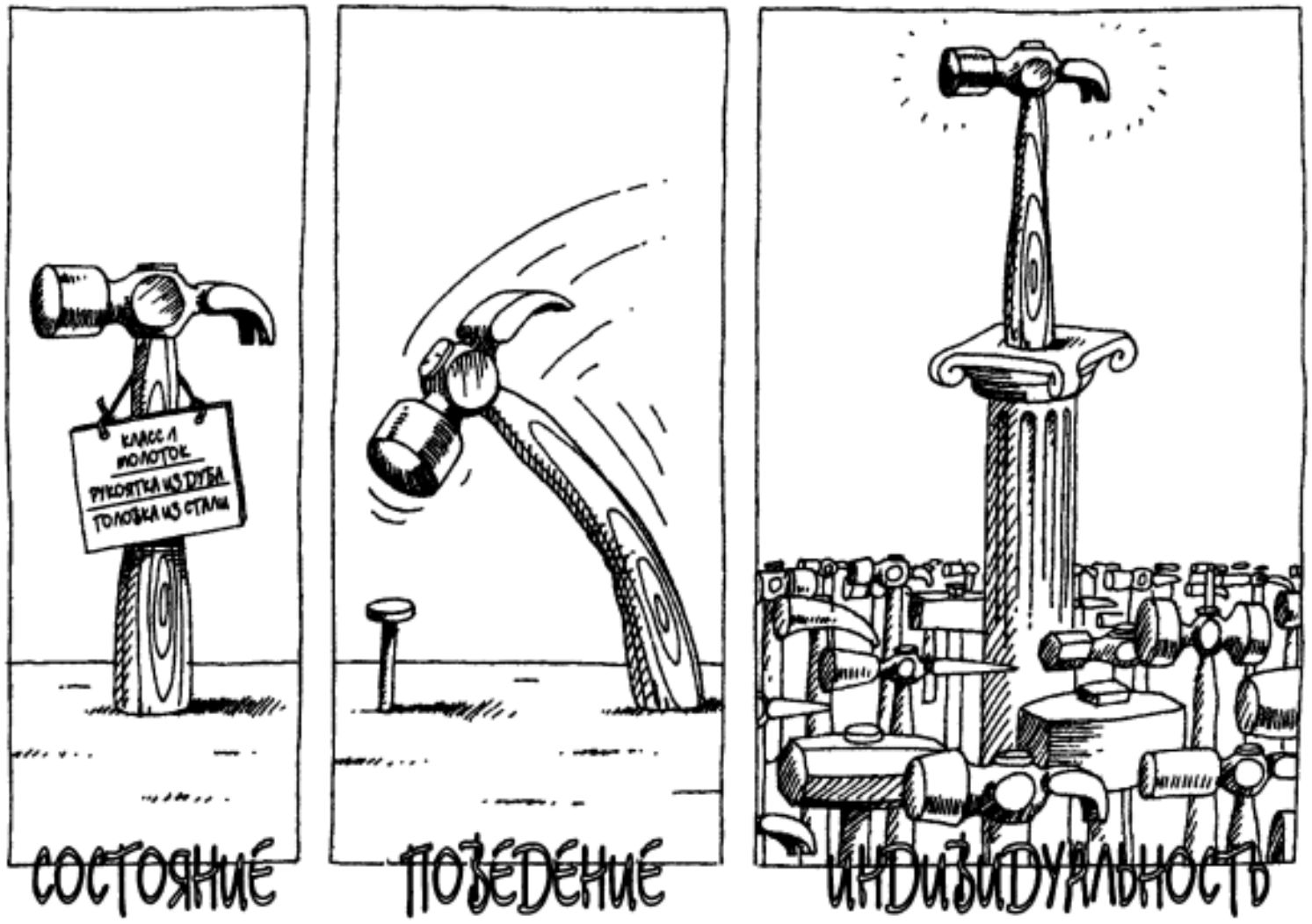
- **Класс** - это шаблон, описание ещё не созданного объекта. Класс содержит данные, которые описывают строение объекта и его возможности, методы работы с ним;

- **Объект** — экземпляр класса. То, что «рождено» по «чертежу», то есть по описанию из класса.

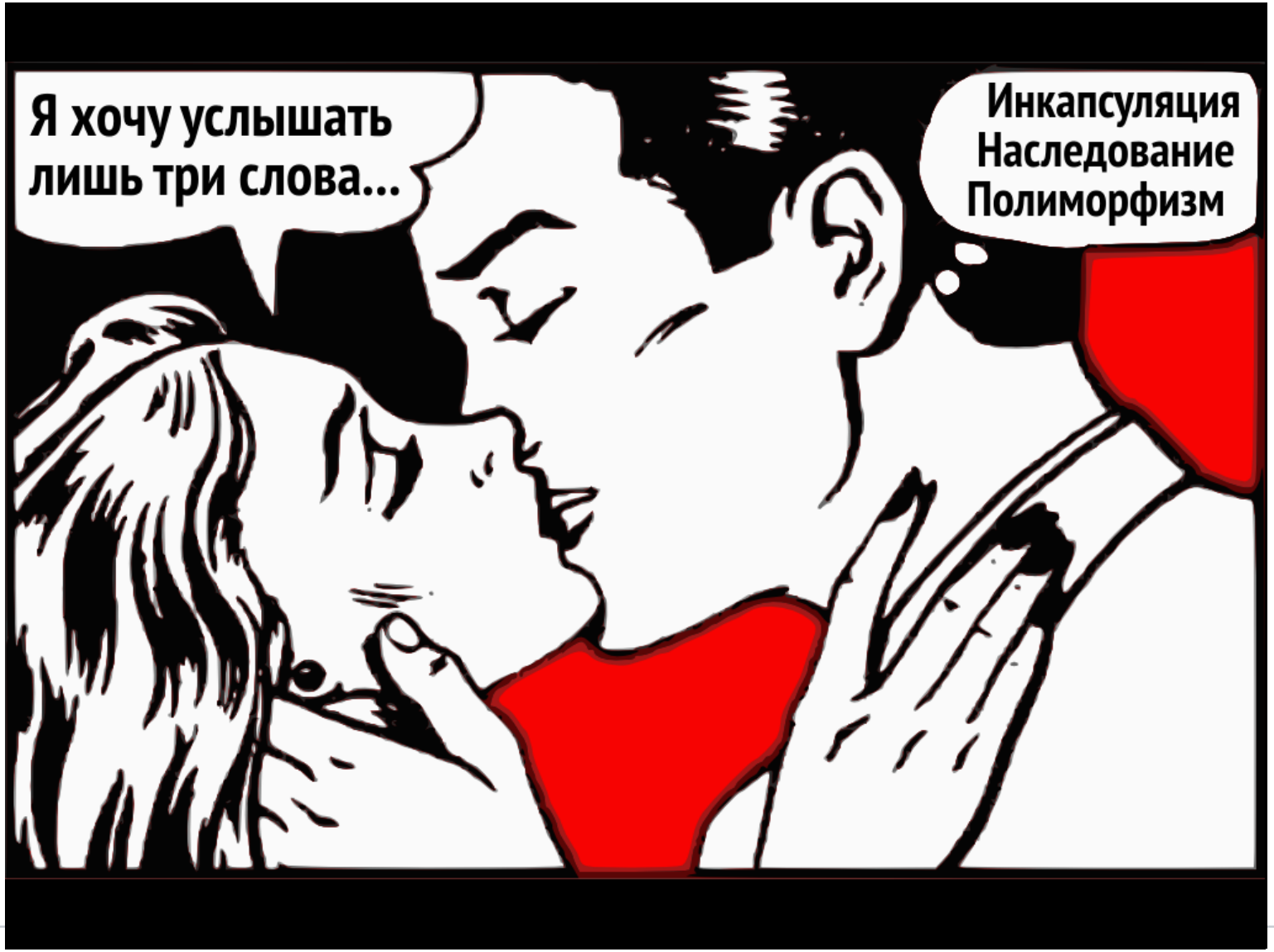
В качестве примера объекта и класса можно привести технический чертёж для изготовления детали — это класс. Выточенная же на станке по размерам и указаниям из чертежа деталь - объект.



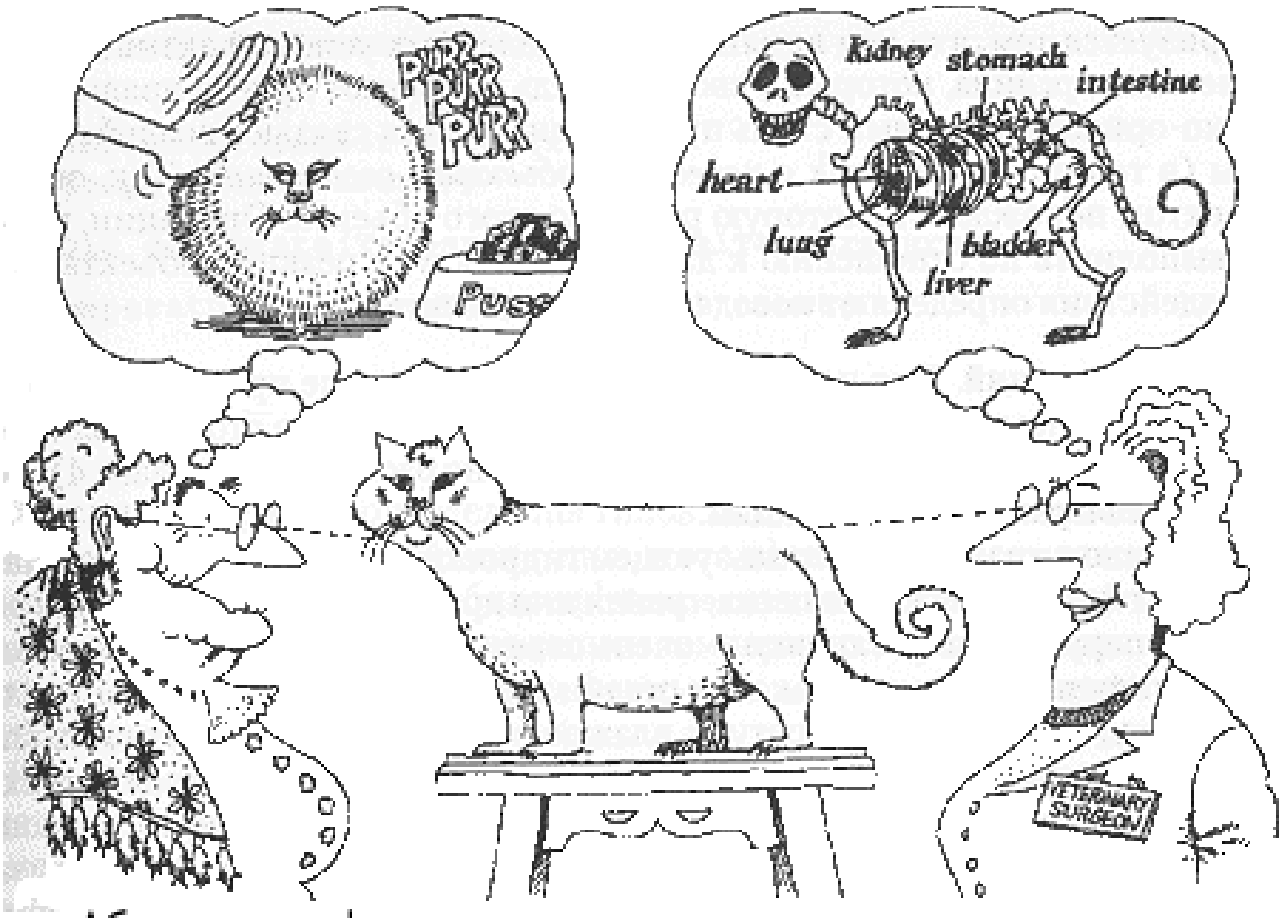
Объекты: состояние, поведение, уникальность



Основные понятия ООП

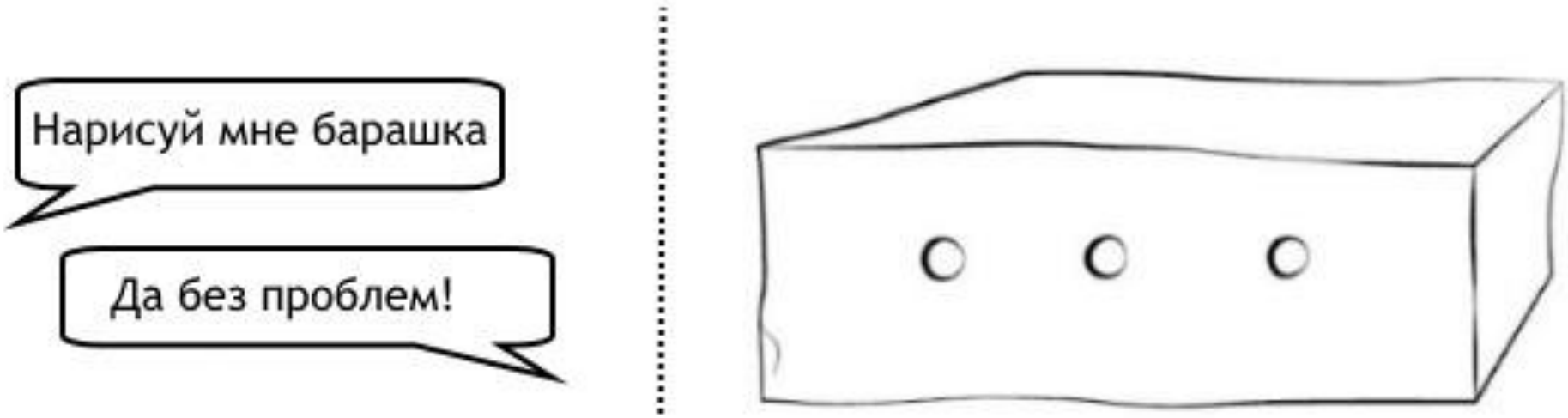


Абстракция



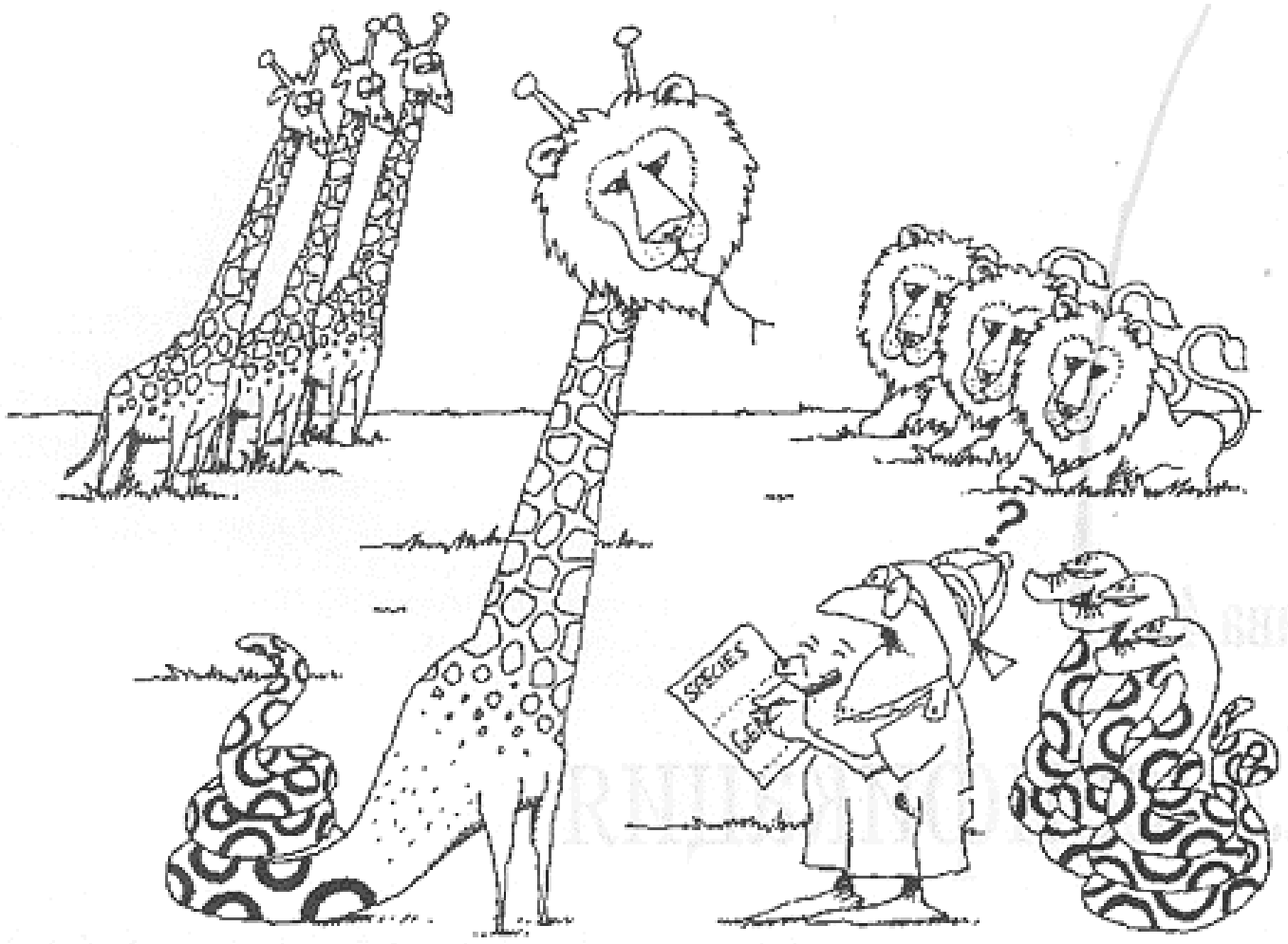
Абстракция фокусирует внимание на существенных характеристиках объекта с точки зрения наблюдателя

Инкапсуляция



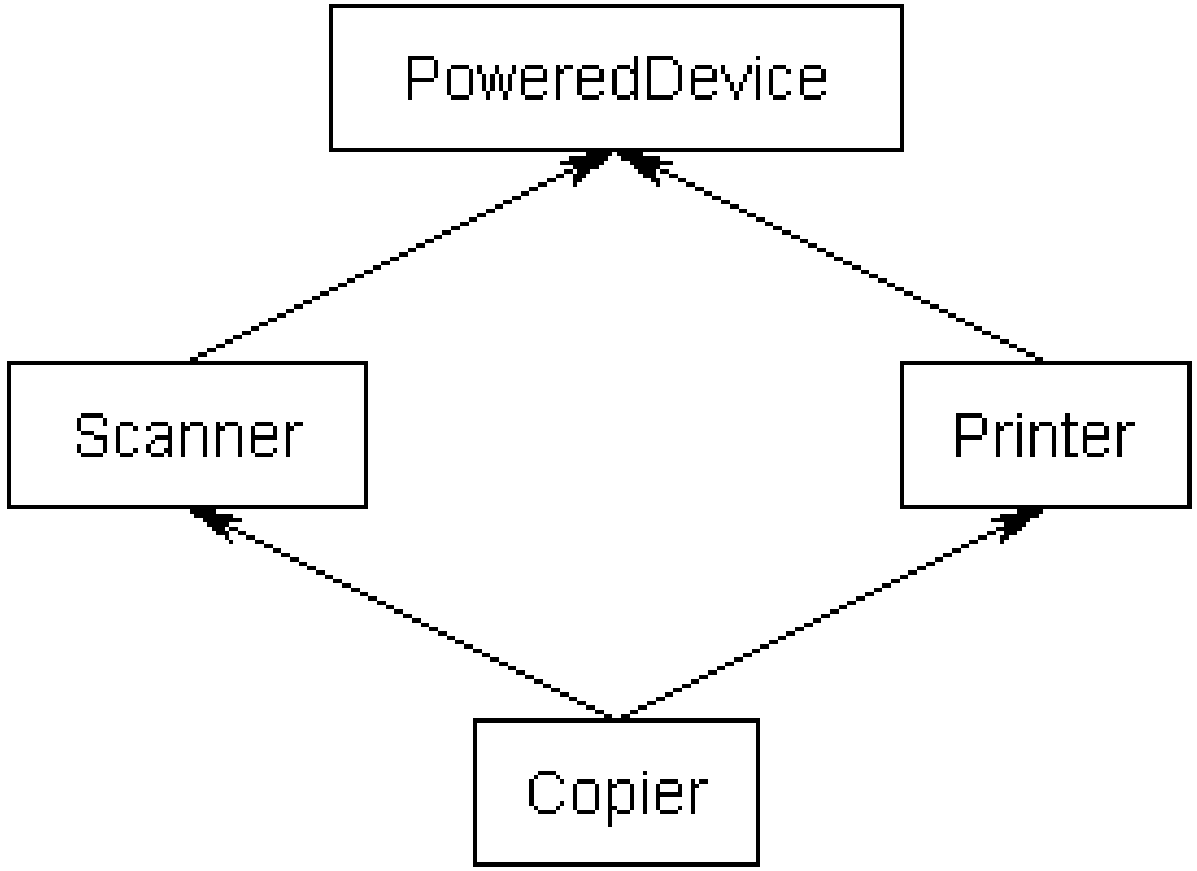
Инкапсуляция позволяет скрывать детали реализации

Наследование



Наследование позволяет приобретать свойства родителей

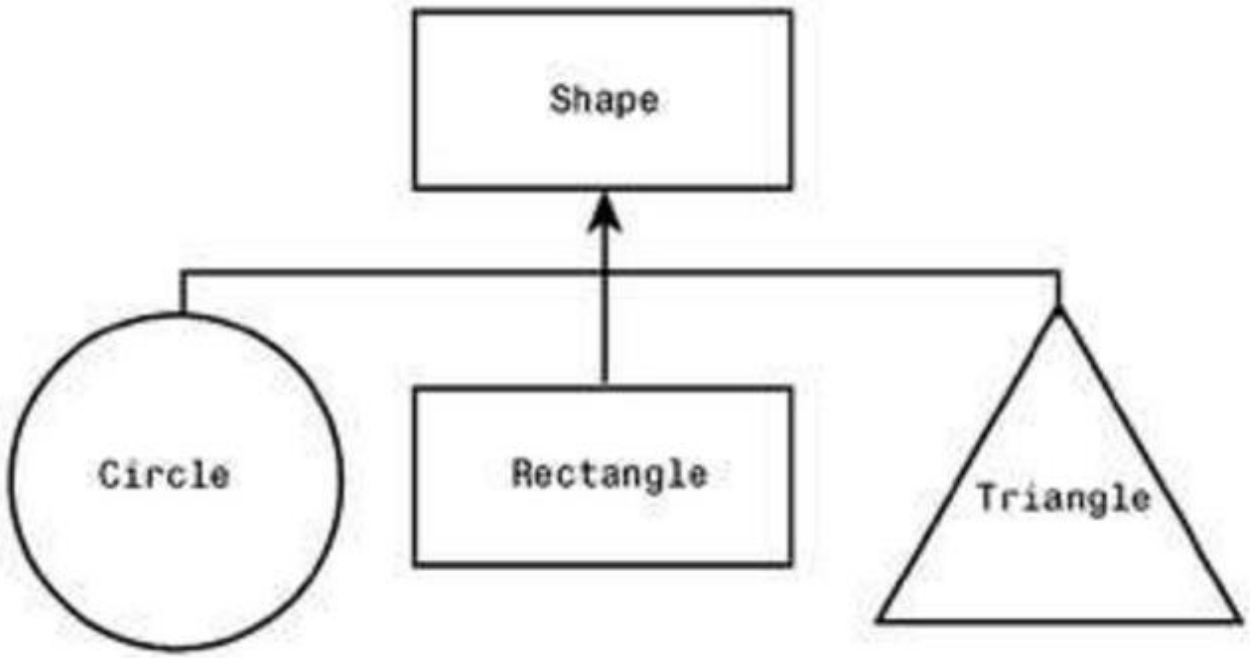
Множественное наследование. Проблема ромба



Полиморфизм

Полиморфизм - это один интерфейс для множества реализаций

Пример



Достоинства ООП

- Классы позволяют проводить конструирование из полезных компонент, обладающих простыми инструментами, что дает возможность абстрагироваться от деталей реализации.
- Данные и операции над ними вместе образуют определенную сущность, и они не разносятся по всей программе, как это нередко бывает в случае процедурного программирования, а описываются вместе. Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.
- Инкапсуляция позволяет привнести свойство модульности, что облегчает распараллеливание выполнения задачи между несколькими исполнителями и обновление версий отдельных компонент.
- ООП дает возможность создавать расширяемые системы. Это одно из самых значительных достоинств ООП, и именно оно отличает данный подход от традиционных методов программирования. Расширяемость означает, что существующую систему можно заставить работать с новыми компонентами, причем без внесения в нее каких либо изменений. Компоненты могут быть добавлены на этапе исполнения программы.
- Обработка разнородных структур данных. Программы могут работать, не различая вида объектов, что существенно упрощает код. Новые виды могут быть добавлены в любой момент.
- Изменение поведения во время исполнения. На этапе исполнения один объект может быть заменен другим, что позволяет легко без изменения кода адаптировать алгоритм, в зависимости от того, какой используется объект.

Достоинства ООП

- Реализация работы с наследниками. Алгоритмы можно обобщить настолько, что они уже смогут работать более чем с одним видом объектов.
- Создание “каркаса” (framework). Независимые от приложения части предметной области могут быть реализованы в виде набора универсальных классов, или каркаса (framework), и в дальнейшем расширены за счет добавления частей, специфичных для конкретного приложения. Часто на практике многоразового использования программного обеспечения добиться не удается из-за того, что существующие компоненты уже не отвечают новым требованиям. ООП помогает этого достичь без нарушения работы уже имеющихся клиентов, что позволяет нам извлечь максимум из многоразового использования компонент.
- Сокращается время на разработку, которое с выгодой может быть отдано другим задачам.
- Компоненты многоразового использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке.
- Когда некая компонента используется сразу несколькими клиентами, то улучшения, вносимые в ее код, одновременно оказывают свое положительное влияние и на множество работающих с ней программ.
- Если программа опирается на стандартные компоненты, то ее структура и пользовательский интерфейс становятся более унифицированными, что облегчает ее понимание и упрощает ее использование.

Недостатки ООП

- Документирование классов - задача более трудная, чем это было в случае процедур и модулей. Поскольку любой метод может быть переопределен, в документации должно говориться не только о том, что делает данный метод, но также и о том, в каком контексте он вызывается. Ведь переопределенные методы обычно вызываются не клиентом, а самим каркасом. Таким образом, программист должен знать, какие условия выполняются, когда вызывается данный метод. Для абстрактных методов, которые пусты, в документации должно даже говориться о том, для каких целей предполагается использовать переопределяемый метод.
- В сложных иерархиях классов поля и методы обычно наследуются с разных уровней. И не всегда легко определить, какие поля и методы фактически относятся к данному классу. Для получения такой информации нужны специальные инструменты вроде навигаторов классов. Если конкретный класс расширяется, то каждый метод обычно сокращают перед передачей сообщения базовому классу. Реализация операции, таким образом, рассредоточивается по нескольким классам, и чтобы понять, как она работает, нам приходится внимательно просматривать весь код.
- Методы, как правило, короче процедур, поскольку они осуществляют только одну операцию над данными. Зато количество методов намного выше. Короткие методы обладают тем преимуществом, что в них легче разбираться, неудобство же их связано с тем, что код для обработки сообщения иногда "размазан" по многим маленьким методам.

Недостатки ООП

- Абстракцией данных не следует злоупотреблять. Чем больше данных скрыто в недрах класса, тем сложнее его расширять. Отправной точкой здесь должно быть не то, что клиентам не разрешается знать о тех или иных данных, а то, что клиентам для работы с классом этих данных знать не требуется.
- Часто можно слышать, что ООП является неэффективным. Как же дело обстоит в действительности? Мы должны четко проводить грань между неэффективностью на этапе выполнения, неэффективностью в смысле распределения памяти и неэффективностью, связанной с излишней универсализацией.
- 1. Неэффективность на этапе выполнения. В языках типа Smalltalk сообщения интерпретируются во время выполнения программы путем осуществления поиска их в одной или нескольких таблицах и за счет выбора подходящего метода. Конечно, это медленный процесс. И даже при использовании наилучших методов оптимизации
  Smalltalk-программы в десять раз медленнее оптимизированных С-программ. В гибридных языках типа Oberon-2, Object Pascal и C++ посылка сообщения приводит лишь к вызову через указатель процедурной переменной. На некоторых машинах сообщения выполняются лишь на 10% медленнее, чем обычные процедурные вызовы. И поскольку сообщения встречаются в программе гораздо реже других операций, их воздействие на время выполнения влияния практически не оказывает. Однако существует другой фактор, который влияет на время выполнения: это инкапсуляция данных. Рекомендуется не предоставлять прямой доступ к полям класса, а выполнять каждую операцию над данными через методы. Такая схема приводит к необходимости выполнения процедурного вызова при каждом доступе к данным. Однако, когда инкапсуляция используется только там, где она необходима (т.е. вслучаях, где это становится преимуществом), то замедление вполне приемлемое.

Недостатки ООП

- 2. Неэффективность в смысле распределения памяти.

Динамическое связывание и проверка типа на этапе выполнения требуют по ходу работы информации о типе объекта. Такая информация хранится в дескрипторе типа, и он выделяется один на класс. Каждый объект имеет невидимый указатель на дескриптор типа для своего класса. Таким образом, в объектно-ориентированных программах требуемая дополнительная память выражается в одном указателе для объекта и в одном дескрипторе типа для класса.

- 3. Излишняя универсальность.

Неэффективность может также означать, что программа имеет ненужные возможности. В библиотечном классе часто содержится больше методов, чем это реально необходимо. А поскольку лишние методы не могут быть удалены, то они становятся мертвым грузом. Это не воздействует на время выполнения, но влияет на возрастание размера кода. Одно из возможных решений - строить базовый класс с минимальным числом методов, а затем уже реализовывать различные расширения этого класса, которые позволят нарастить функциональность. Другой подход - дать возможность компоновщику удалять лишние методы. Такие интеллектуальные компоновщики уже доступны для различных языков и операционных систем. Но нельзя утверждать, что ООП неэффективно. Если классы используются лишь там, где это действительно необходимо, то потеря эффективности из-за повышенного расхода памяти и меньшей производительности незначительна. Кроме того, часто более важной является надежность программного обеспечения и небольшое время его написания, а не производительность .

Литература

- Ильдар Хабибуллин, Самоучитель java.
- Брюс Эккель, Философия java.
- Гради Буч, Объектно-ориентированный анализ и проектирование с примерами приложений.
- https://devcolibri.com/%D1%87%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-%D0%BE%D0%BE%D0%BF-%D0%B8-%D1%81-%D1%87%D0%B5%D0%BC-%D0%B5%D0%B3%D0%BE-%D0%B5%D0%B4%D1%8F%D1%82/

Q&A

Focused forward

Netcracker

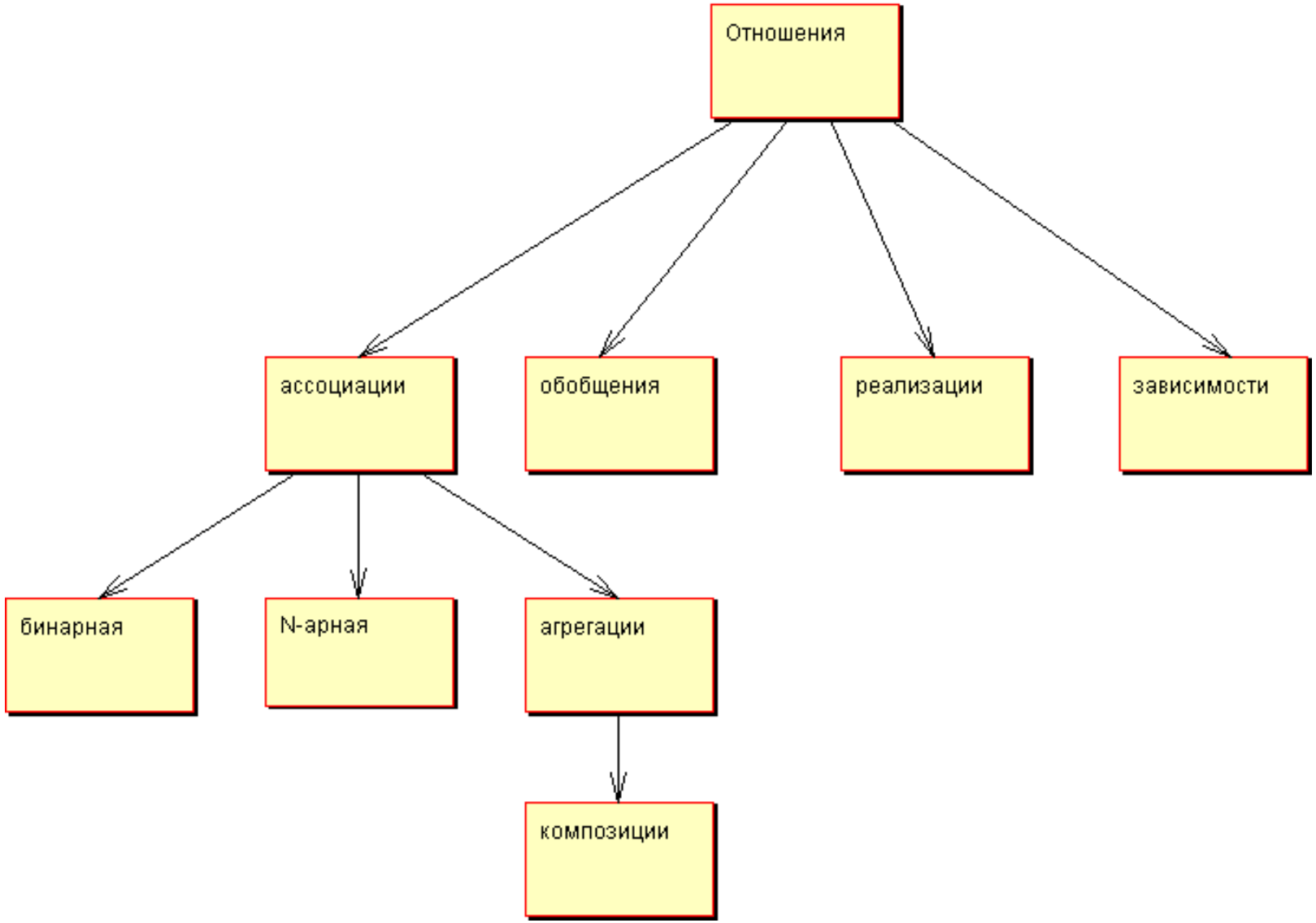
Lecture 1. Part 2. UML

Alexei Khudnitsky

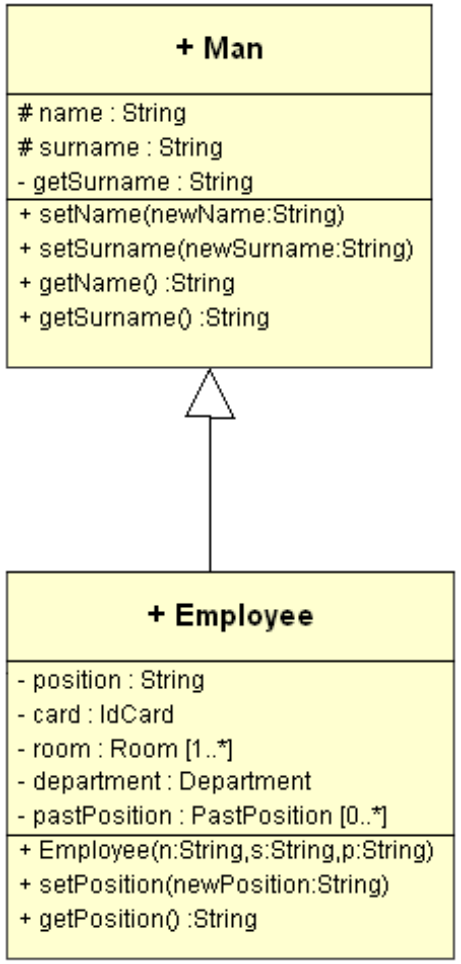
Цель

- построить UML-диаграмму классов (Class Model), а затем отразить ее в объектно-ориентированном коде.
- прикладная область - отдел кадров некого предприятия.

Отношения классов



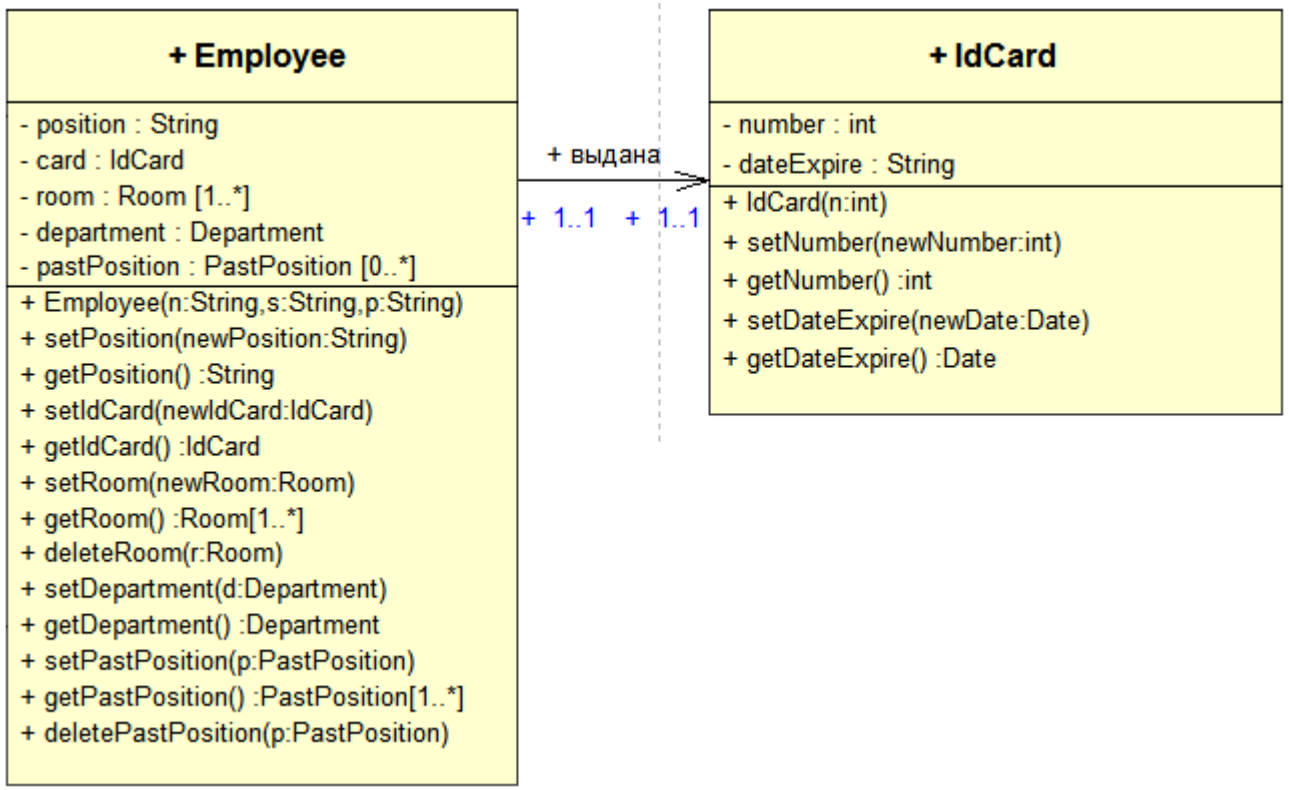
Обобщение




```java
public class Man {

    protected String name;
    protected String surname;
}
```

```java
public class Employee extends Man {

    // 1. Обобщение
    private String position;
```

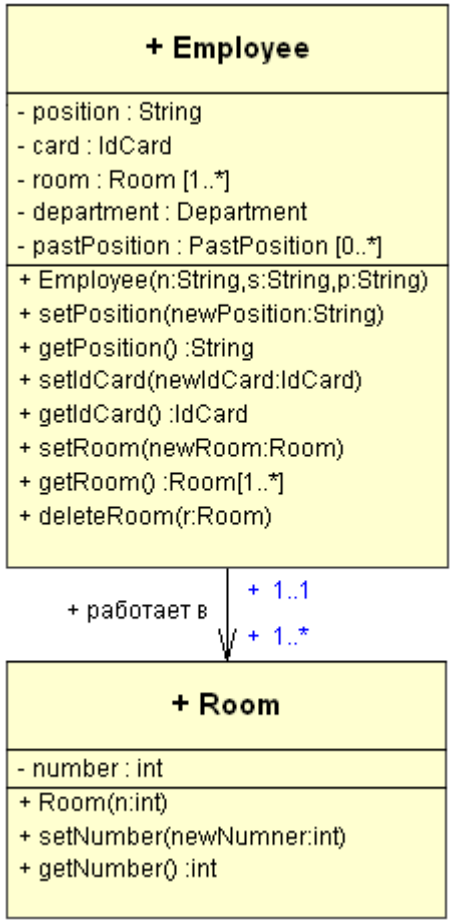
Ассоциация (бинарная)




```java
//2. Бинарная ассоциация
public class IdCard {
    private Date dateExpire;
    private int number;
```

```java
public class Employee extends Man {

    // 1. Обобщение
    private String position;

    //2. Бинарная ассоциация
    private IdCard idCard;
```

Ассоциация (n-арная)



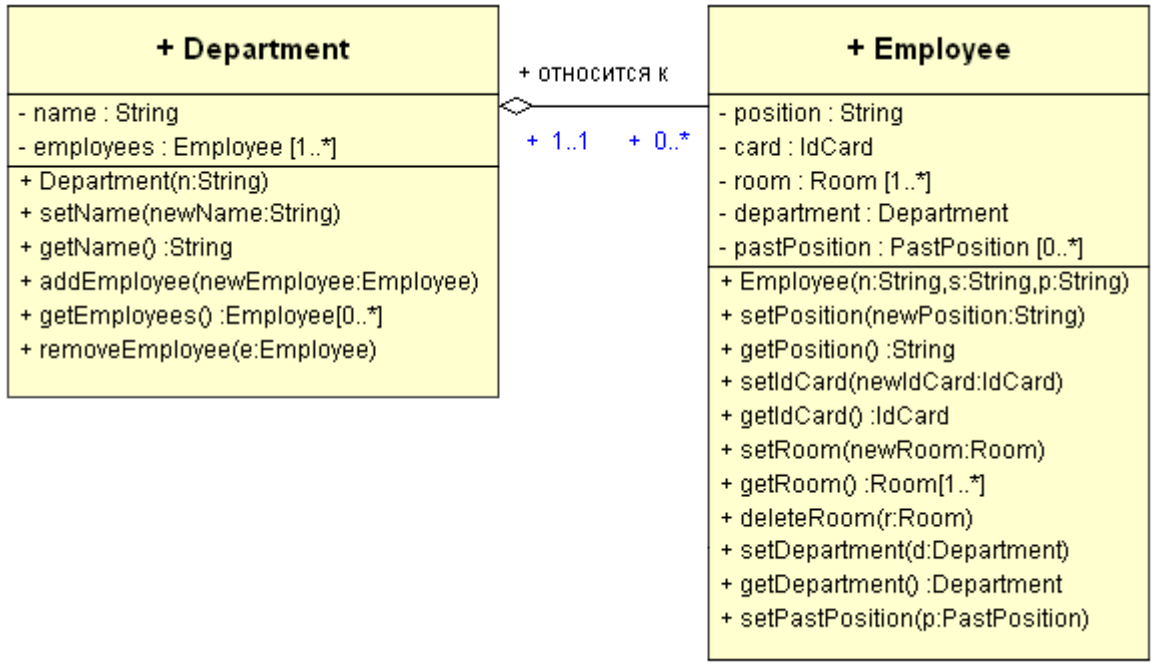

```java
public class Employee extends Man {

    // 1. Обобщение
    private String position;

    //2. Бинарная ассоциация
    private IdCard idCard;

    // 3. N-арная ассоциация
    private Set rooms = new HashSet();
```

```java
// 3. N-арная ассоциация
public class Room {
    private int number;
```

Ассоциация

- Если объекты одного класса ссылаются на один или более объектов другого класса, но ни в ту, ни в другую сторону отношение между объектами не носит характера "владения« или контейнеризации, то такое отношение называют ассоциацией (association).
- Отношение ассоциации изображается так же,как и отношение агрегации, но линия, связывающая классы - простая, без ромбика.

Агрегация



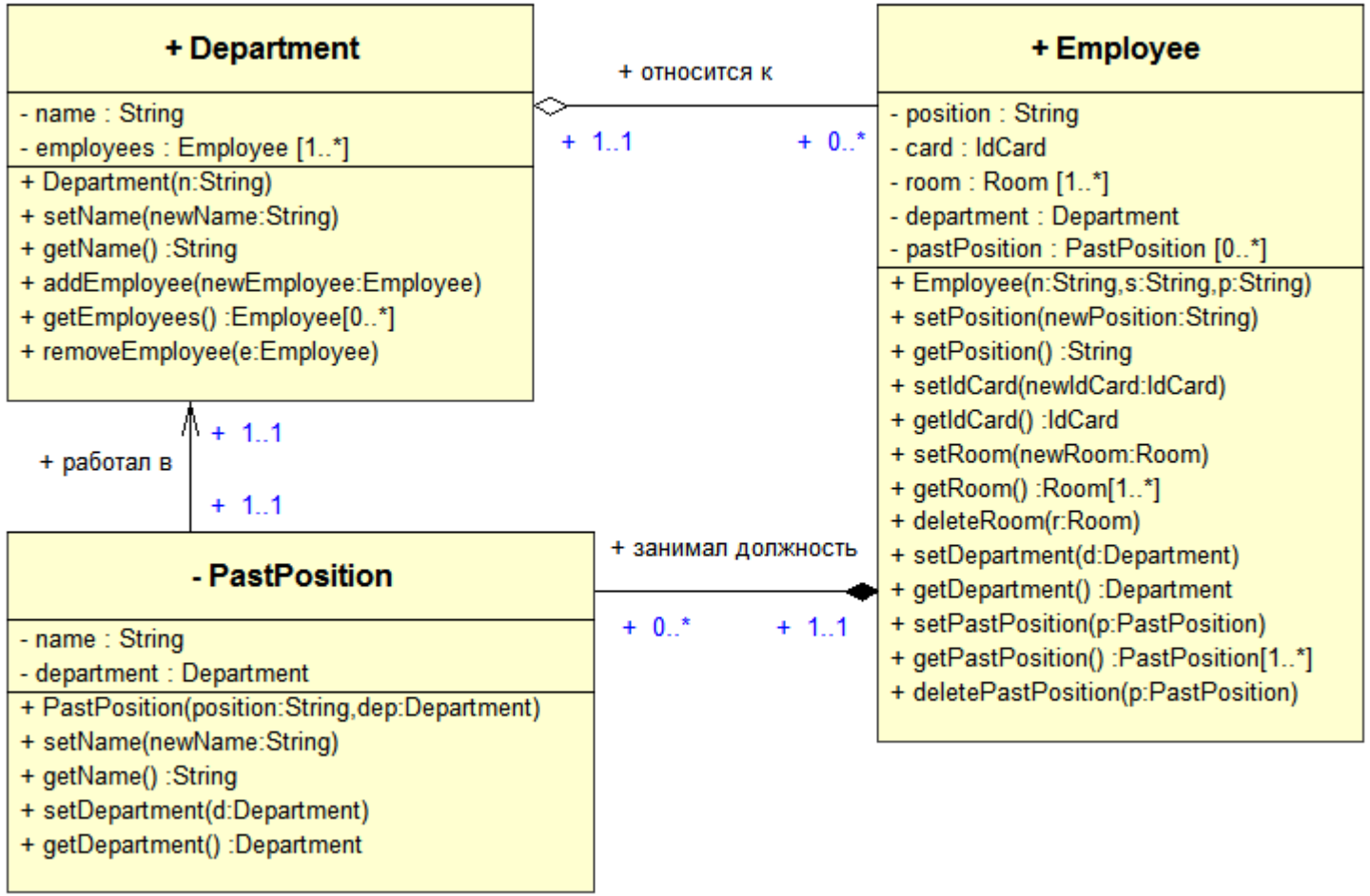

```java
//4. Агрегация
public class Department implements Unit{
    private String name;
    private Set<Employee> employees = new HashSet<>();

    public void addEmployee(Employee newEmployee){
        employees.add(newEmployee);
        // связываем сотрудника с этим отделом
        newEmployee.setDepartment(this);
    }

    public void removeEmployee(Employee e) { employees.remove(e); }
```

```java
public class Employee extends Man {

    // 1. Обобщение
    private String position;

    //2. Бинарная ассоциация
    private IdCard idCard;

    // 3. N-арная ассоциация
    private Set rooms = new HashSet();

    //4. Агрегация
    private Department department;
```

Агрегация

- Отношение между классами типа "содержит" или "состоит из" называется агрегацией или включением. Например, если аквариум наполнен водой и в нем плавают рыбки, то можно сказать, что аквариум агрегирует в себе воду и рыбок.
- Такое отношение включения или агрегации (aggregation) изображается линией с ромбиком на стороне того класса, который выступает в качестве владельца или контейнера.
- Необязательное название отношения записывается посередине линии.
- В нашем примере отношение "contain" является двунаправленным. Объект класса Aquarium содержит несколько объектов Fish. В то же время каждая рыбка "знает", в каком именно аквариуме она живет. Факт участия класса в отношении изображается посредством роли.
- В примере можно видеть роль "home" класса Aquarium (аквариум является домом для рыбок), а также роль "inhabitants" класса Fish (рыбки являются обитателями аквариума).
- Название роли обычно совпадает с названием соответствующего поля в классе.
- Изображение такого поля на диаграмме излишне, если уже изображено имя роли. Т.е. в данном случае класс Aquarium будет иметь свойство (поле) inhabitants, а класс Fish - свойство home.

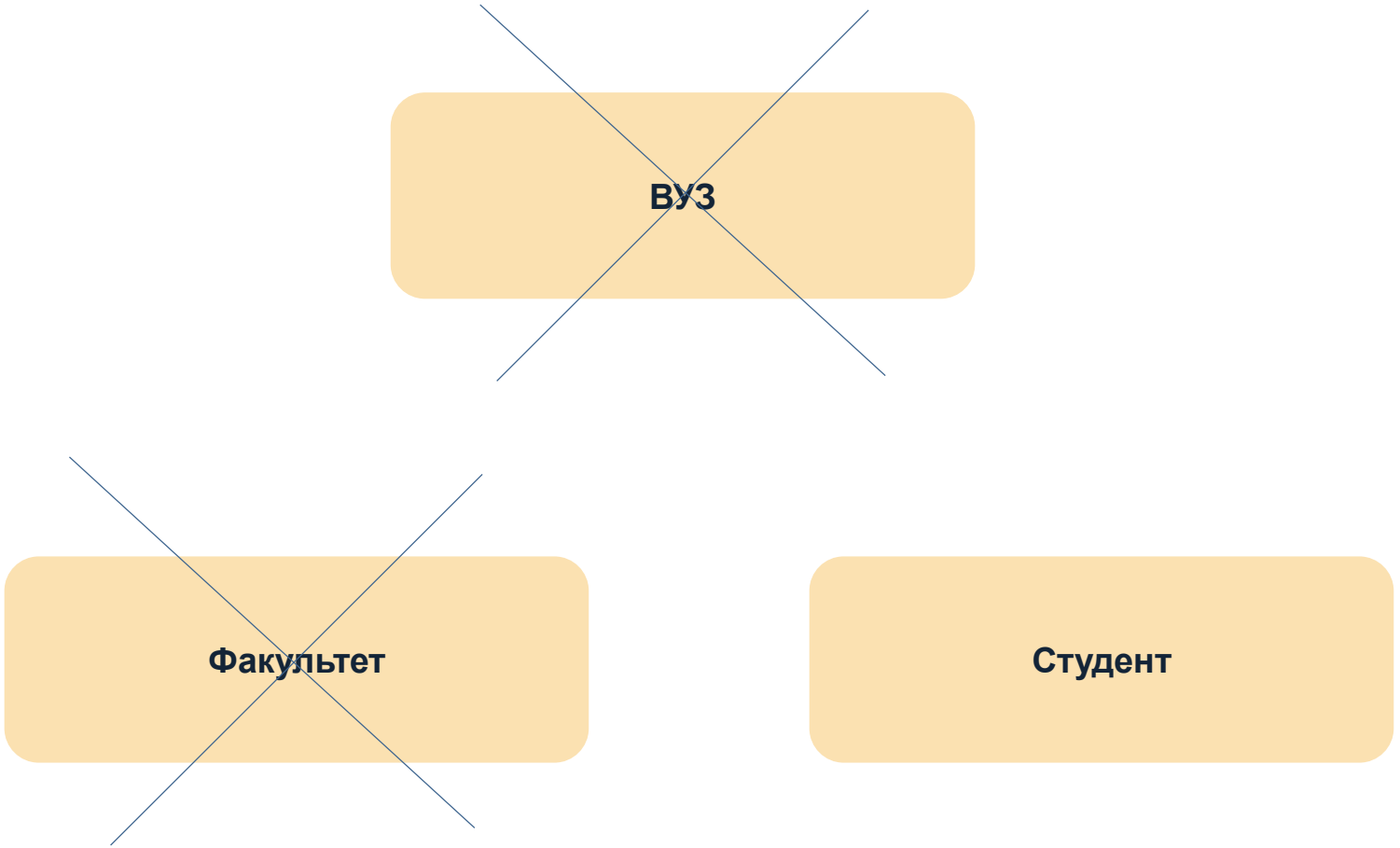
Композиция



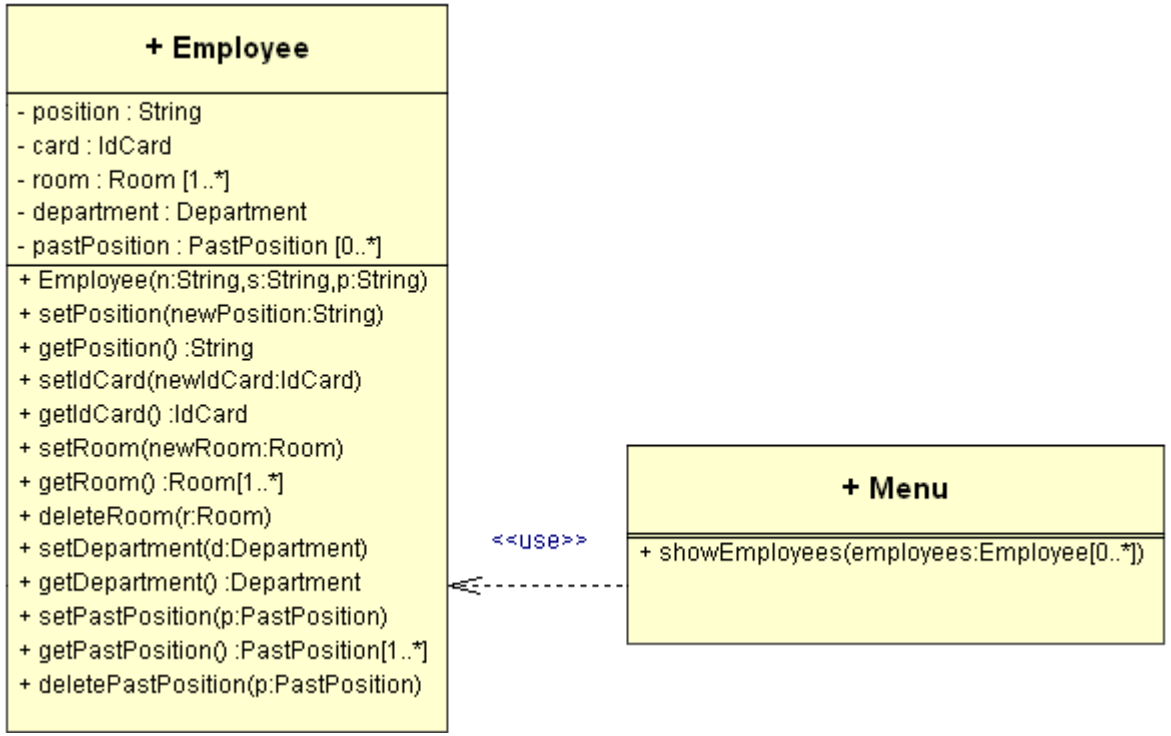

```java
public class Employee extends Man {

    // 1. Обобщение
    private String position;

    //2. Бинарная ассоциация
    private IdCard idCard;

    // 3. N-арная ассоциация
    private Set rooms = new HashSet();

    //4. Агрегация
    private Department department;

    //5. Композиция
    private Set pastPosition = new HashSet();
```

```java
//5. Композиция
public class PastPosition {
    private String name;
    private Department department;
```

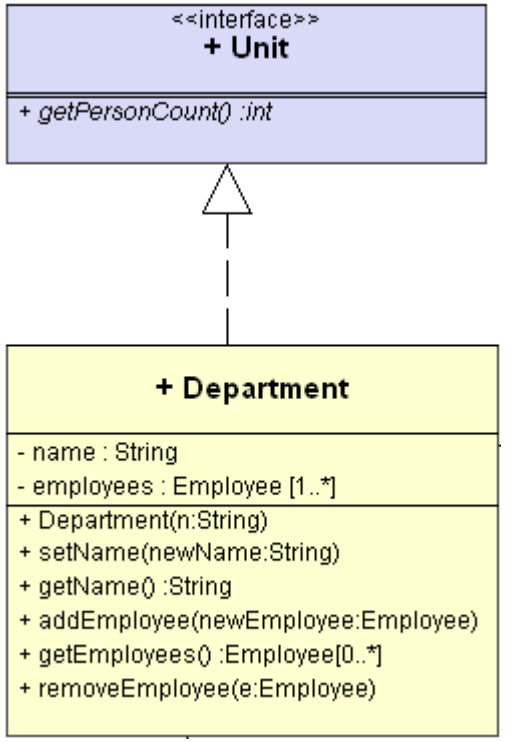
Композиция vs. Агрегация



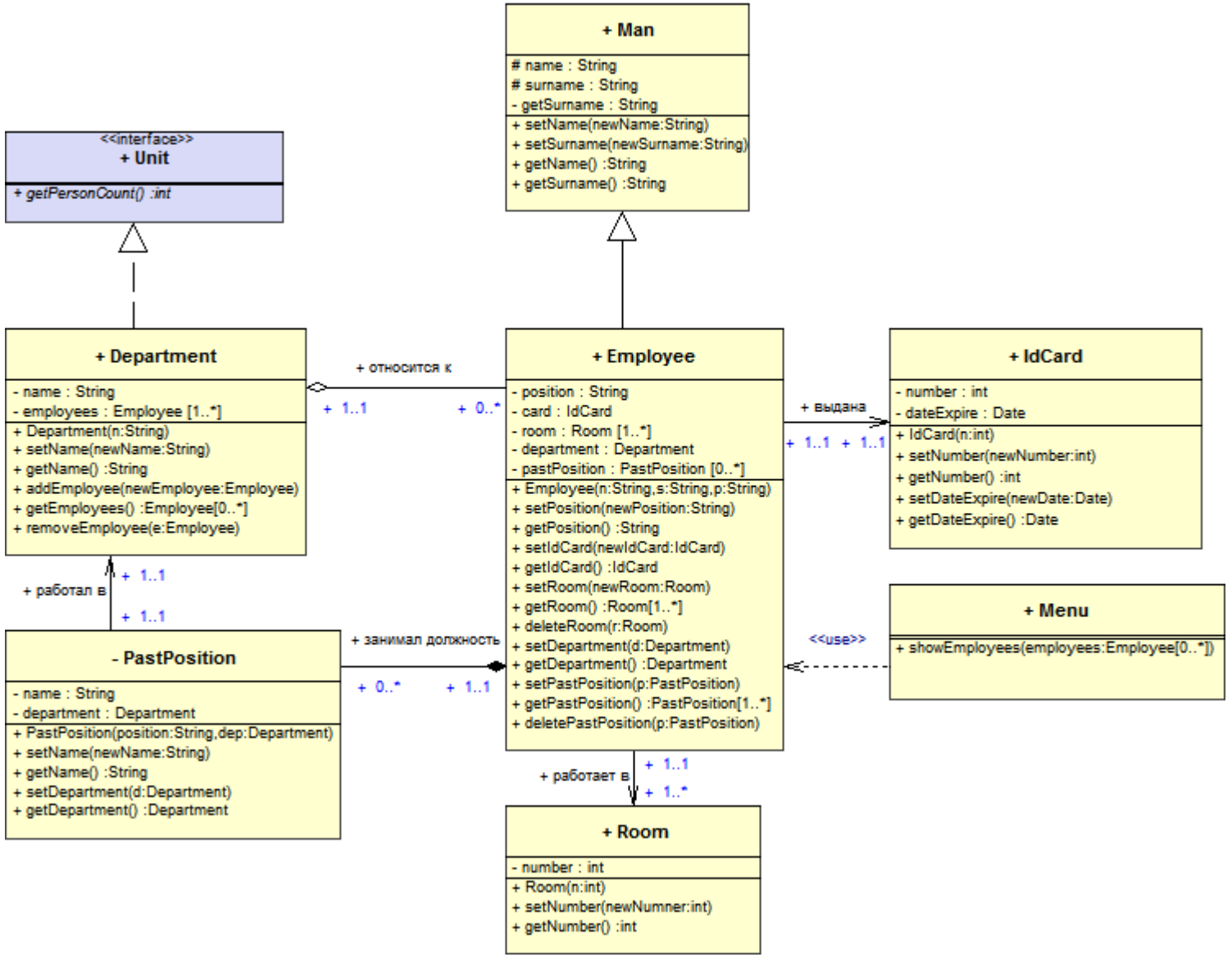
Зависимость



Реализация


<<interface>>
+ Unit
+ getPersonCount() :int
+ Department
- name : String
- employees : Employee [1..*]
+ Department(n:String)
+ setName(newName:String)
+ getName() :String
+ addEmployee(newEmployee:Employee)
+ getEmployees() :Employee[0..*]
+ removeEmployee(e:Employee)

Итог


+ Man
# name : String
# surname : String
- getSurname : String
+ setName(newName:String)
+ setSurname(newSurname:String)
+ getName() :String
+ getSurname() :String
<<interface>>
+ Unit
+ getPersonCount() :int
+ Department
- name : String
- employees : Employee [1..*]
+ Department(n:String)
+ setName(newName:String)
+ getName() :String
+ addEmployee(newEmployee:Employee)
+ getEmployees() :Employee[0..*]
+ removeEmployee(e:Employee)
+ относится к
+ 1..1
+ 0..*
+ Employee
- position : String
- card : IdCard
- room : Room [1..*]
- department : Department
- pastPosition : PastPosition [0..*]
+ Employee(n:String,s:String,p:String)
+ setPosition(newPosition:String)
+ getPosition() :String
+ setIdCard(newIdCard:IdCard)
+ getIdCard() :IdCard
+ setRoom(newRoom:Room)
+ getRoom() :Room[1..*]
+ deleteRoom(r:Room)
+ setDepartment(d:Department)
+ getDepartment() :Department
+ setPastPosition(p:PastPosition)
+ getPastPosition() :PastPosition[1..*]
+ deletePastPosition(p:PastPosition)
+ выдана
+ 1..1
+ 1..1
+ IdCard
- number : int
- dateExpire : Date
+ IdCard(n:int)
+ setNumber(newNumber:int)
+ getNumber() :int
+ setDateExpire(newDate:Date)
+ getDateExpire() :Date
+ работал в
+ 1..1
+ 1..1
- PastPosition
- name : String
- department : Department
+ PastPosition(position:String,dep:Department)
+ setName(newName:String)
+ getName() :String
+ setDepartment(d:Department)
+ getDepartment() :Department
+ занимал должность
+ 0..*
+ 1..1
<<use>>
+ Menu
+ showEmployees(employees:Employee[0..*])
+ работает в
+ 1..1
+ 1..*
+ Room
- number : int
+ Room(n:int)
+ setNumber(newNumner:int)
+ getNumber() :int

Литература

- Г. Буч, Д. Рамбо, А. Джекобсон. Язык UML Руководство пользователя.
- А.В. Леоненков. Самоучитель UML
- Эккель Б. Философия Java. Библиотека программиста. — СПб: Питер, 2001. — 880 с.

Q&A

Focused forward

Netcracker

Lecture 1. Part 3. SOLID Principles

Alexei Khudnitsky

Автор SOLID

Роберт Мартин (Uncle Bob)

сформулировал пять основных принципов объектно-ориентированного программирования и проектирования.

Признаки плохого проекта

- Закрепощённость
- Неустойчивость
- Неподвижность
- Вязкость
- Неоправданная сложность
- Неопределенность

Цель SOLID

Проектировать модули, которые:

- способствуют изменениям
- легко понимаемы
- повторно используемы

SRP: The Single Responsibility Principle

A module should be responsible to one, and only one, actor.

Старая формулировка: ***A module should have one, and only one, reason to change*.**

Каждый класс выполняет лишь одну задачу

OCP: The Open Closed Principle

A software artifact should be open for extension but closed for modification.
Старая формулировка: ***You should be able to extend a classes behavior, without modifying it***.

«программные сущности … должны быть открыты для расширения, но закрыты для модификации.»

LSP: The Liskov Substitution Principle

Derived classes must be substitutable for their base classes

Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

ISP: The Interface Segregation Principle

Make fine grained interfaces that are client specific.

«много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения.»

DIP: The Dependency Inversion Principle

Depend on abstractions, not on concretions.

«Зависимость на Абстракциях.
Нет зависимости на что-то конкретное.»

Common Design Principles

Литература

- https://blog.byndyu.ru/2009/10/solid.html
- https://habr.com/post/343966/

Q&A

Thank You