

# Язык UML

## Кодогенерация на основе UML описаний

Лапутенко Андрей Владимирович  
Евтушенко Нина Владимировна

# План

## 1 Основы UML

### 1.1 Типы диаграмм

#### 1.1.1 Диаграмма классов (Class diagram)

#### 1.1.2 Диаграмма состояний (State machine diagram)

## 2 Кодогенерация на языке Java в среде Visual Paradigm

## 3 Переход от UML диаграмм к автоматной модели для генерации тестов для полученного кода

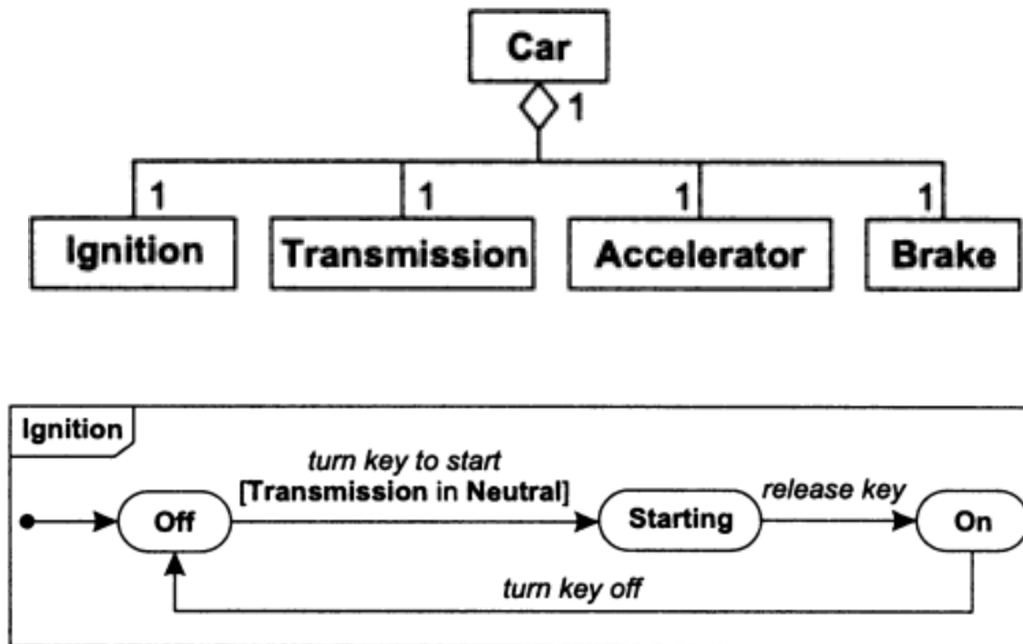
# Unified Modelling Language

## Унифицированный язык моделирования

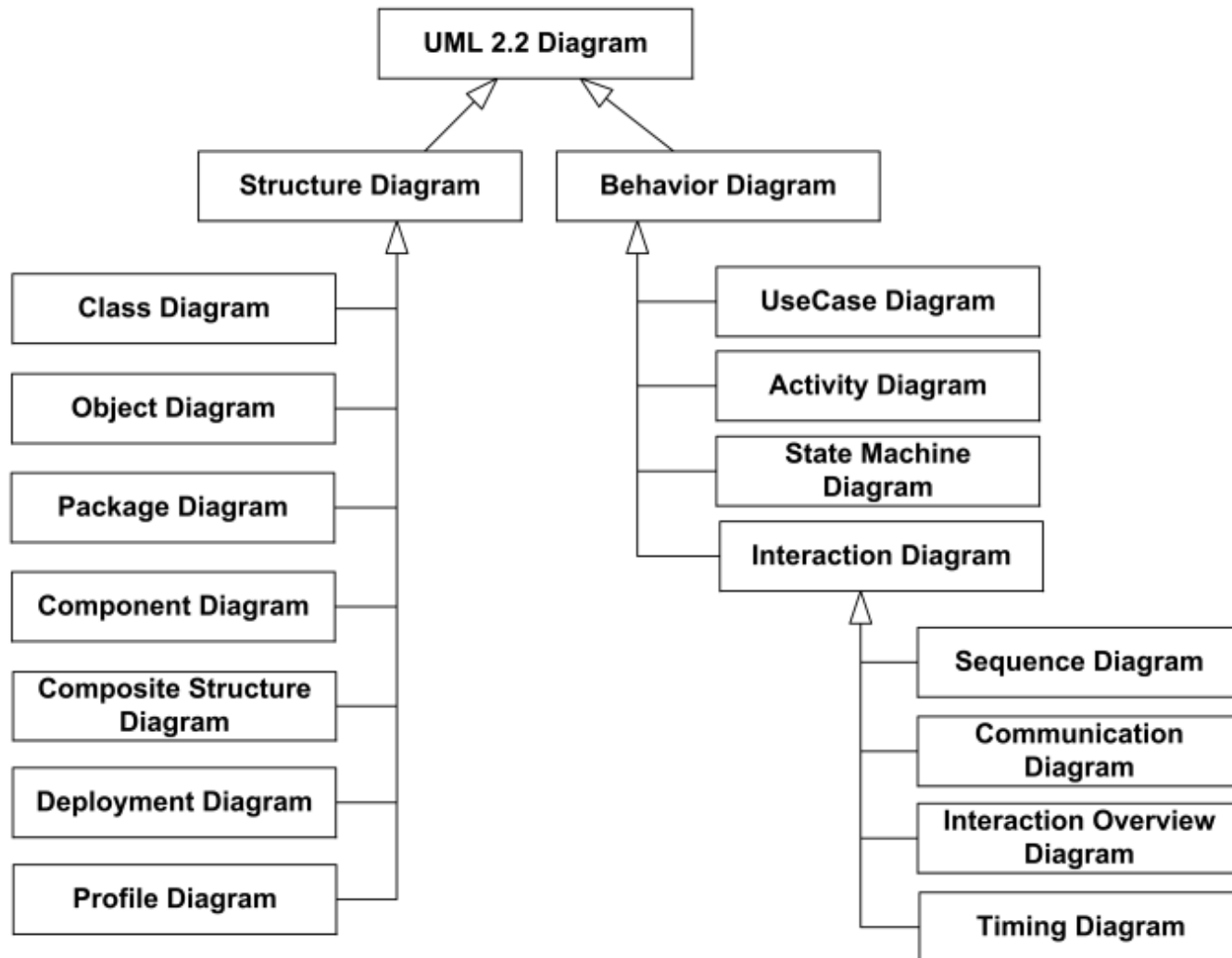
- Язык UML стал объединением трех методов с их доработкой
- Метод Booch (Grady Booch)
- Метод Object Modeling Technique (James Rumbaugh)
- Метод Object Oriented Software Engineering (Ivar Jacobson)
- 1995г – UML 0.8
- Сферы применения
  - информационные системы масштаба предприятия
  - банковские и финансовые услуги
  - телекоммуникации
  - транспорт
  - оборонная промышленность, авиация и космонавтика
  - медицинская электроника
  - распределенные Web-системы

# Определение диаграммы

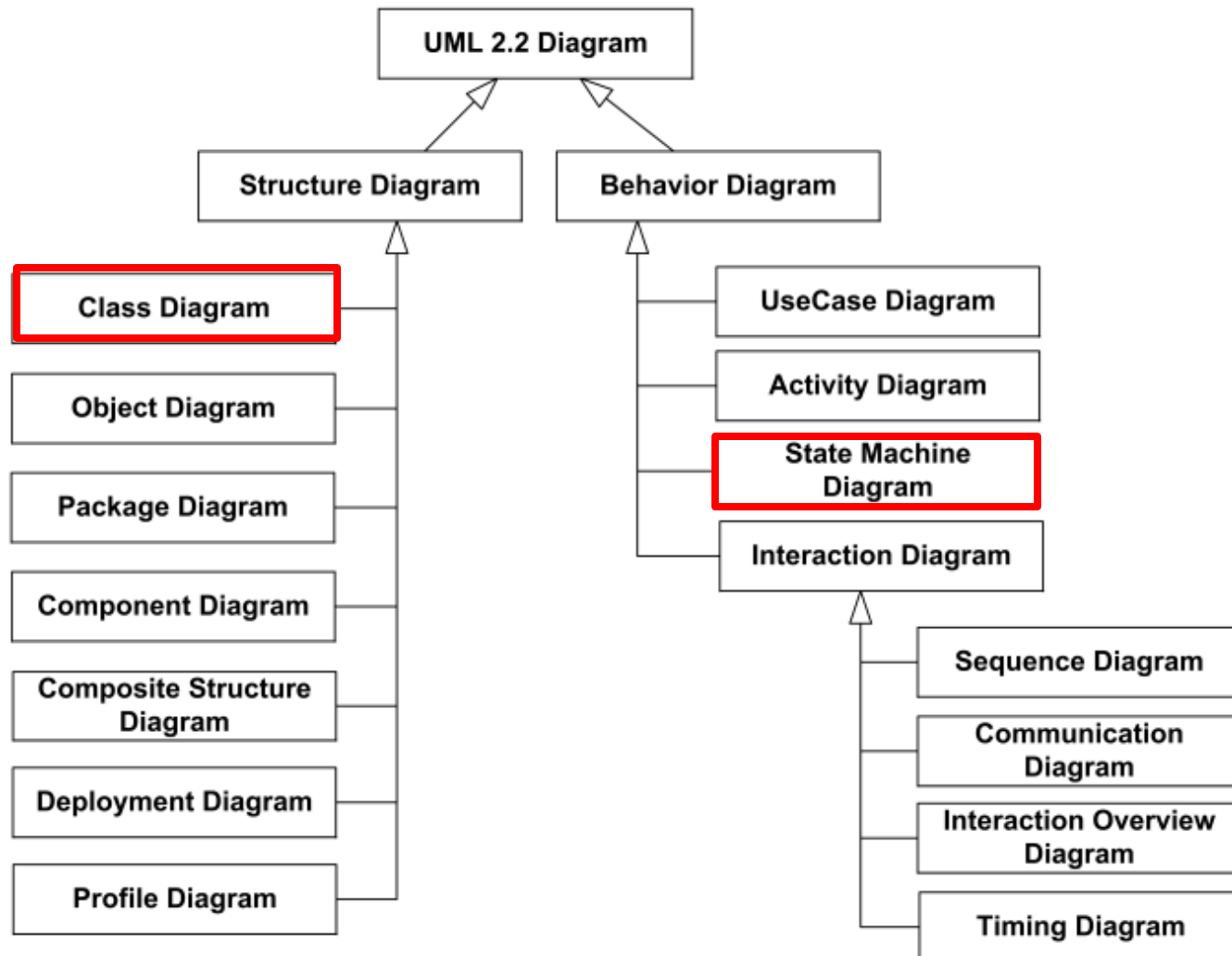
Диаграмма UML – графическое представление структурной или поведенческой информации о проектируемой системе



# Иерархия диаграмм UML 2.2

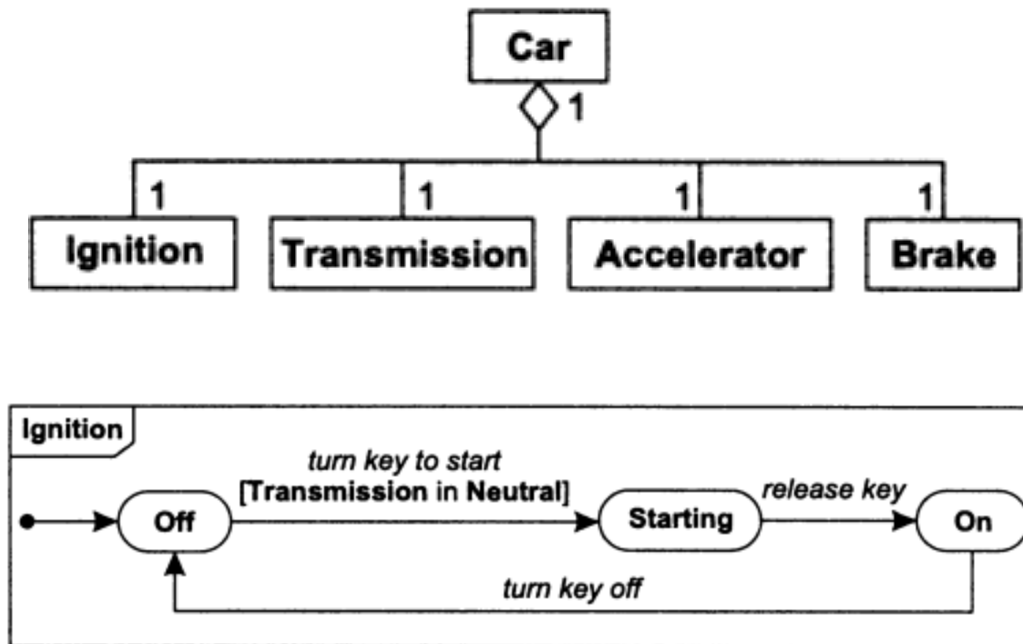


# Иерархия диаграмм UML 2.2



# Определение диаграммы

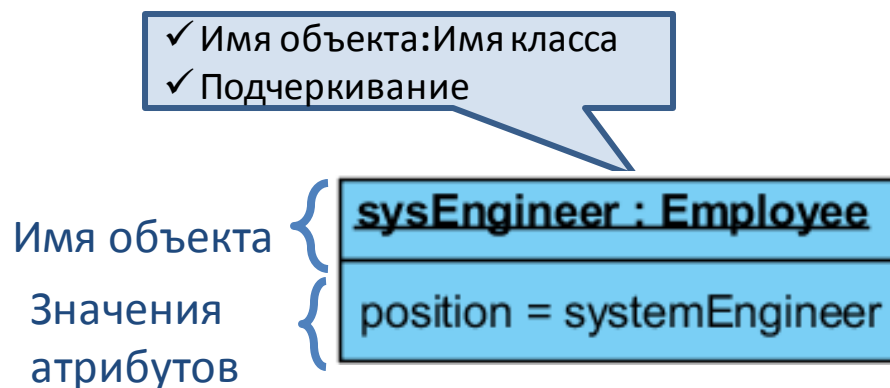
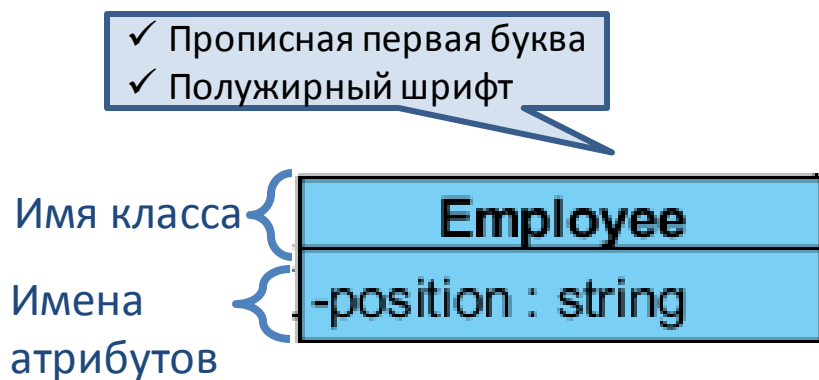
Диаграмма UML – графическое представление структурной или поведенческой информации о проектируемой системе



# Классы и объекты

Класс описывает группу объектов с одинаковыми свойствами (атрибутами), одинаковым поведением (операциями), типами отношений и семантикой\*

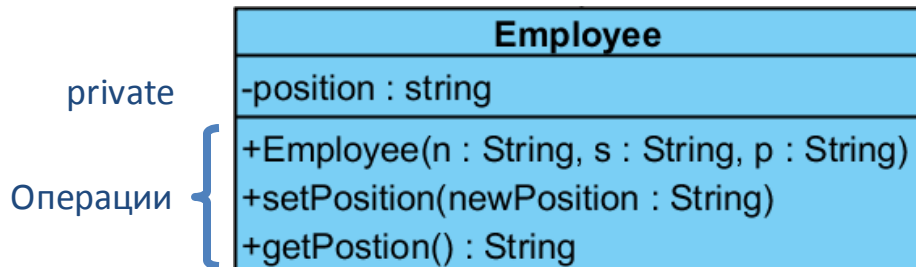
Объект – концепция, абстракция или сущность, обладающая индивидуальностью и имеющая смысл в рамках приложения\*





# Основные понятия

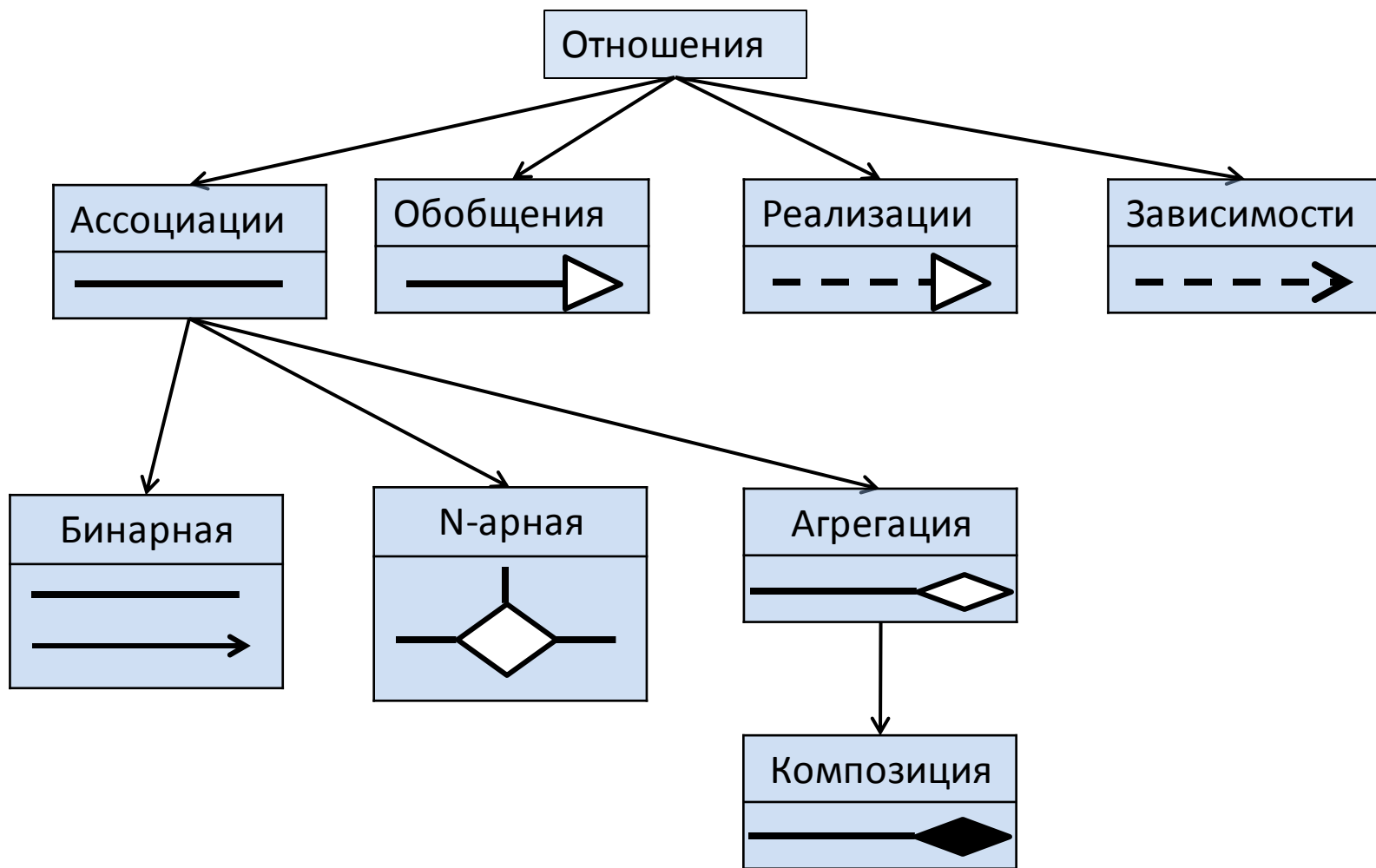
- Атрибут – общее свойство объектов класса
- Операция – функция или процедура, применимая к объектам класса
- Метод – реализация операции в программном коде, конкретный алгоритм



```
public class Employee{  
    private String position;  
    public Employee(String n, String s, String p){  
    }  
    public void setPosition(String newProfession){  
    }  
    public String getPostion(){  
    }  
}
```

операция в UML **==** метод в программном коде

# Отношения между классами на диаграммах

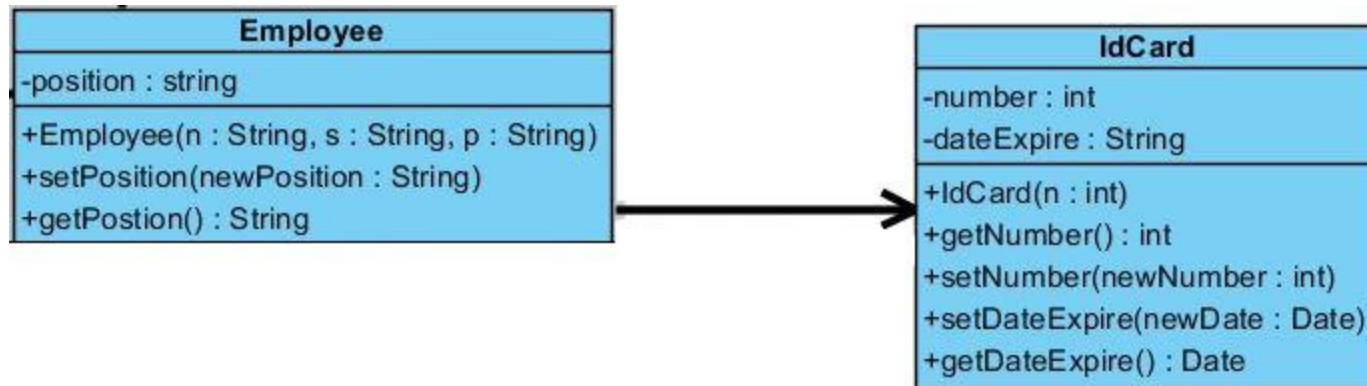


# Отношения на диаграммах

## Ассоциации

Ассоциация указывает на наличие *произвольной* связи между классами. Объекты одного класса связаны с объектами другого класса таким образом, что можно перемещаться от объектов одного класса к другому.

Например: Сотруднику выдается идентификационная карточка



Из объекта типа **Employee** можем получить информацию об объекте типа **IdCard**

# Отношения на диаграммах

## Ассоциации

Класс IdCard

```
public class IdCard{
    private Date dateExpire;
    private int number;
    public IdCard(int n){
        number = n;
    }
    public void setNumber(int newNumber){
        number = newNumber;
    }
    public int getNumber(){
        return number;
    }
    public void setDateExpire(Date newDateExpire){
        dateExpire = newDateExpire;
    }
    public Date getDateExpire(){
        return dateExpire;
    }
}
```

Класс Employee

```
public class Employee {
    private String position;
    private IdCard IndividualCard;

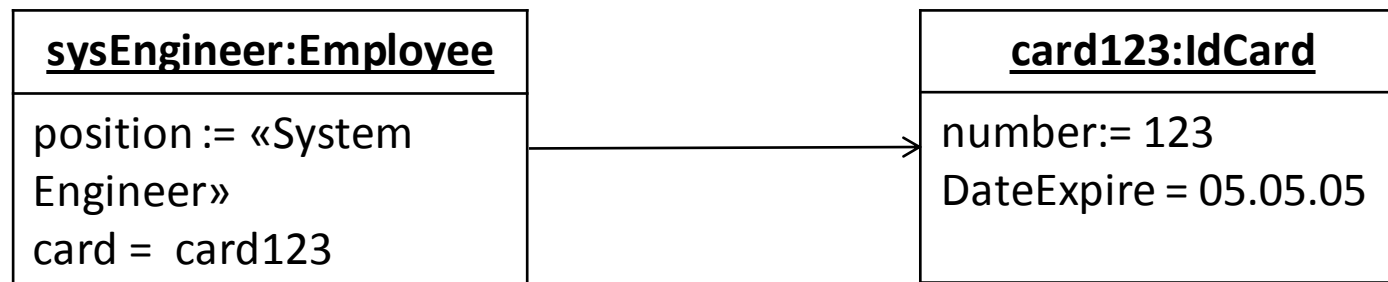
    public Employee(String n, String s, String p){
        name = n;
        surname = s;
        position = p;
    }
    public void setPosition(String newPosition){
        position = newPosition;
    }
    public String getPosition(){
        return position;
    }
    public void setIdCard(IdCard c){
        iCard = c;
    }
    public IdCard getIdCard(){
        return iCard;
    }
}
```

# Отношения на диаграммах

## Ассоциации и связи

Связь – экземпляр ассоциации, связывающий объекты тех классов, которые участвуют в ассоциации

Например: работник на должности системного инженера имеет карту с номером 123



Направление ассоциации помогает понять, что из объекта типа `Employee` мы можем узнать об объекте типа `IdCard`

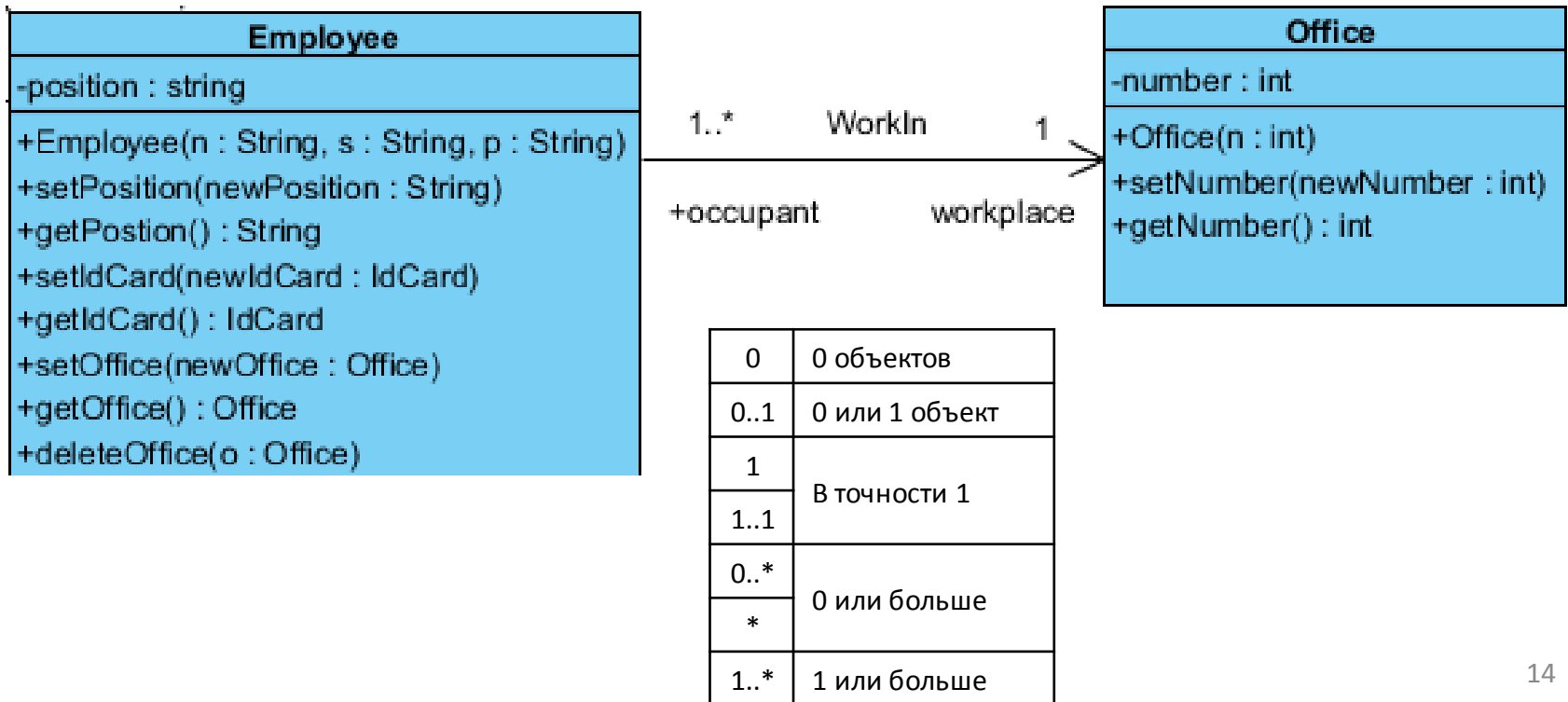
Ассоциация описывает множество потенциальных связей так же, как класс описывает множество потенциальных объектов

# Отношения на диаграммах

## Кратность ассоциации

Кратность – количество экземпляров одного класса, которые могут быть связаны с одним экземпляром другого класса через одну ассоциацию

Например: «В одном помещении может работать множество людей и один человек может работать только в одном помещении»



# Отношения на диаграммах

## Кратность ассоциации

### Класс Office

```
public class Office{
    private int number;
    private Set<Employee> employees = new
HashSet<Employee>();

    public Office (int n){
        number = n;
    }
    public void setNumber(int newNumber){
        number = newNumber;
    }
    public int getNumber(){
        return number;
    }
}
```

### Данный код добавляется в класс Employee

```
...
private Set office= new Office();
...
public void setOffice(Office newOffice){
    Office.add(newOffice);
}
public Set getOffice(){
    return Office;
}
public void deleteOffice (Office r){
    Office.remove(r);
}
...
```

# Отношения на диаграммах

## Имена полюсов ассоциации

Полюс ассоциации – конец ассоциации, связанный с классом

Имя полюса ассоциации – имя роли, которую играет класс в данной ассоциации



На рабочей станции может быть открыто не более одного окна, которое является консолью

Каждое консольное окно может быть открыто только на одной рабочей станции

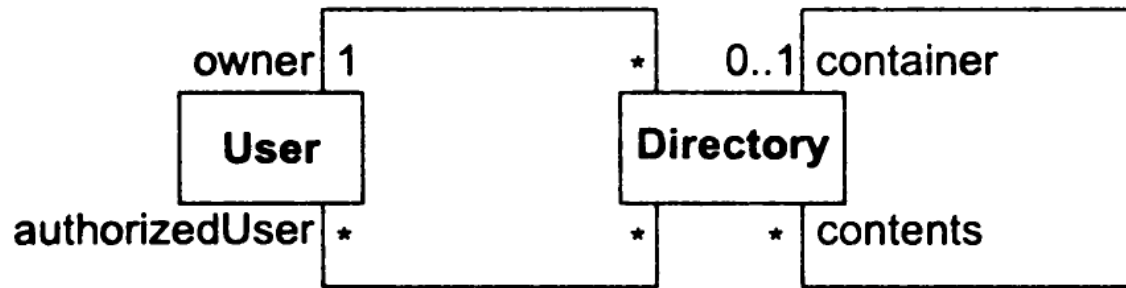


# Отношения на диаграммах

## Ассоциации между объектами одного класса

Имена полюсов ассоциации обязательны при установлении связи между двумя объектами одного класса

Например, каталог может содержать вложенные каталоги и может содержаться в других каталогах



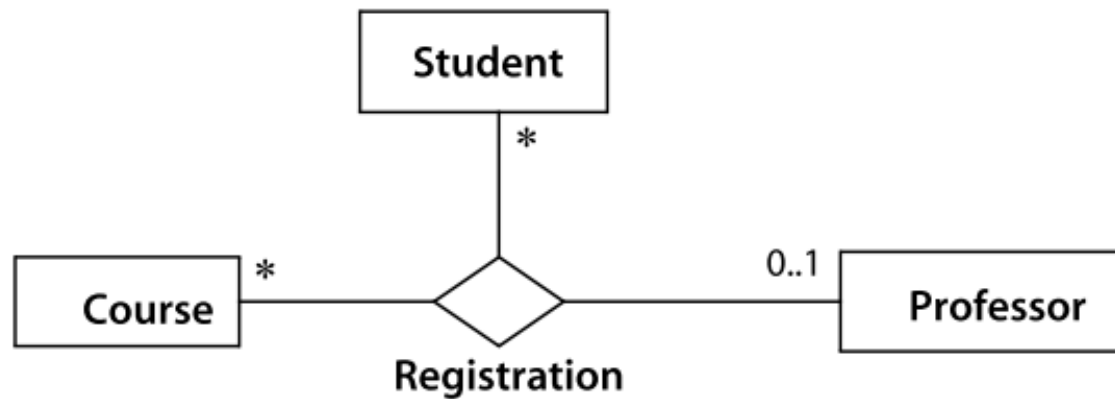
Имена полюсов ассоциации позволяют различать разные ассоциации между одними и теми же классами

Каждый каталог имеет одного пользователя, являющегося владельцем и множество пользователей, которые имеют право работать с каталогом

# Отношения на диаграммах

## N-арная ассоциация

N-арная ассоциация – ассоциация, заданная на множестве из N классов  
Основанием для введения классов ассоциаций послужила возможность создания ассоциаций типа многие ко многим



# Отношения на диаграммах

## N-арная ассоциация

При программной реализации n-арной ассоциации, она представляется классом, содержащим атрибуты – ссылки на все объекты ассоциации

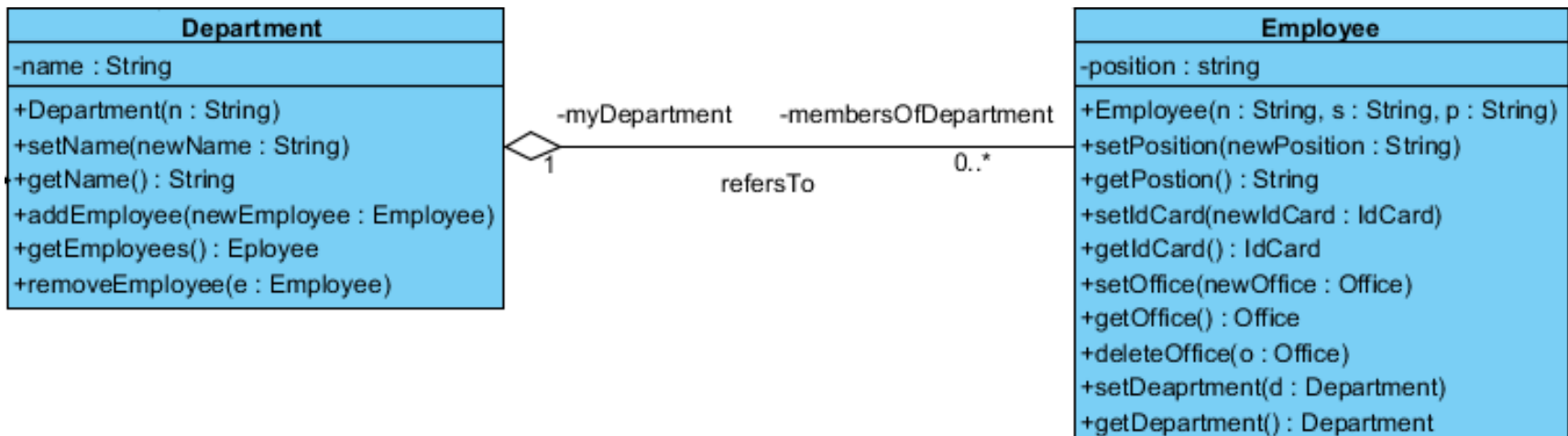
```
public class Registration() {  
    public Student _student;  
    public Course _course;  
    public Professor _professor;  
    .....  
}
```

# Отношение агрегации

Агрегация – частный случай ассоциации, описывающий объекты, состоящие из частей

Агрегация – общая форма отношения «часть - целое»

Например, отдел состоит из работников



Направленности ассоциации нет, значит, об объекте типа **Employee** можно узнать из объекта класса **Department** и наоборот

# Отношение агрегации

## Класс Department

```
public class Department{
    private String name;
    private Set<Employee> employees = new
HashSet<Employee>();
    public Department(String n){
        name = n;
    }
    public void setName(String newName){
        name = newName;
    }
    public String getName(){
        return name;
    }
    public void addEmployee(Employee newEmployee){
        employees.add(newEmployee);
        newEmployee.setDepartment(this);
    }
    public Set getEmployees(){
        return employees;
    }
    public void removeEmployee(Employee e){
        employees.remove(e);
    }
}
```

Данный код добавляется в класс Employee

```
...
private Department department;
...
public void setDepartment(Department d){
    department = d;
}
public Department getDepartment(){
    return department;
}
```

# Отношение композиции

Композиция – частный случай агрегации с двумя дополнительными условиями

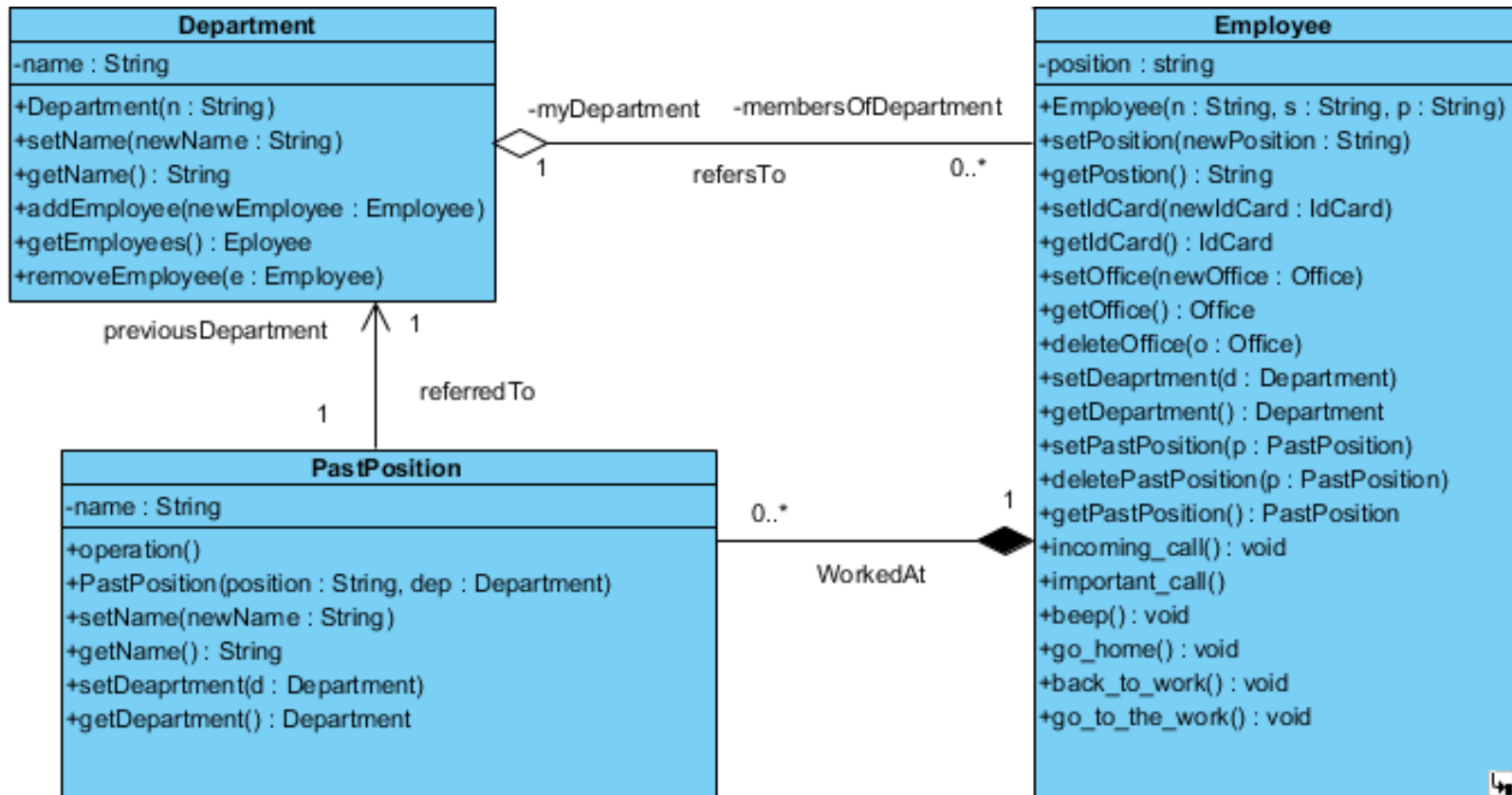
- составляющая часть может принадлежать не более чем одному агрегату
- составляющая часть получает срок жизни, равный сроку жизни агрегата

Композиция – частная форма отношения «часть - целое»

По умолчанию, агрегация является агрегацией по ссылке

Композиция – агрегация по значению

# Отношение композиции



# Отношение композиции

## Класс PastPosition

```
public class PastPosition {  
    private String name;  
    private Department previousDepartment;  
    public Employee workedAt;  
  
    public void operation() {  
        throw new UnsupportedOperationException();  
    }  
  
    public PastPosition(String position, Department dep)  
    {  
        throw new UnsupportedOperationException();  
    }  
}
```

Данный код добавляется в класс Employee

```
...  
public Vector<PastPosition> workedAt = new  
    Vector<PastPosition>();  
...  
public void setPastPosition(PastPosition p){  
    pastPosition.add(p);  
}  
public Set getPastPosition(){  
    return pastPosition;  
}  
public void deletePastPosition(PastPosition p){  
    pastPosition.remove(p);  
}  
...
```



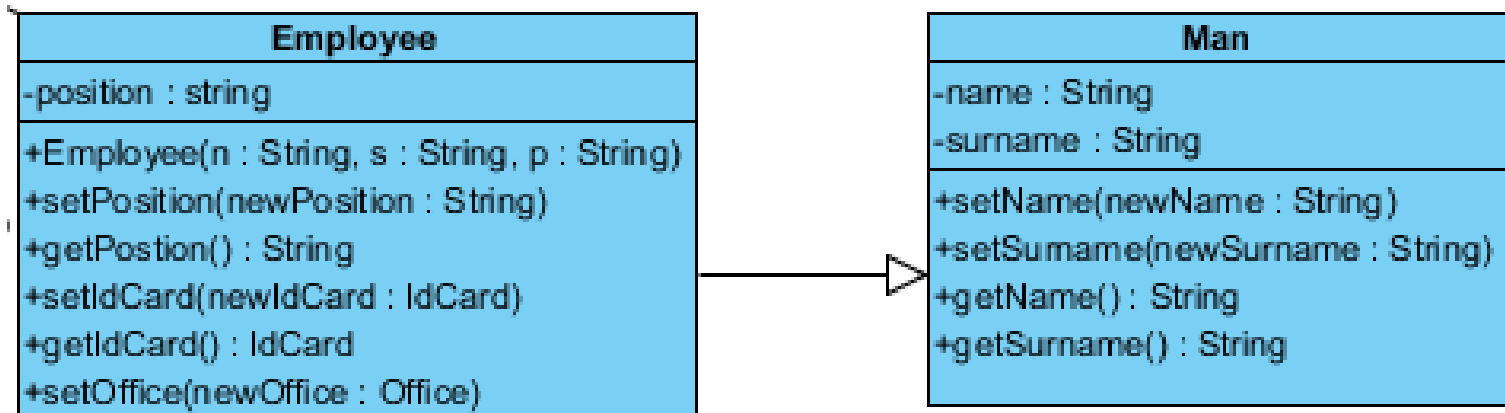
# Отношения на диаграммах

## Обобщение (наследование)

**Человек** – суперкласс, родитель, базовый класс (тип)

**Работник** – подкласс, потомок, наследованный класс (типы)

Родитель и потомок находятся в отношении «является»



# Отношения на диаграммах

## Обобщение (наследование)

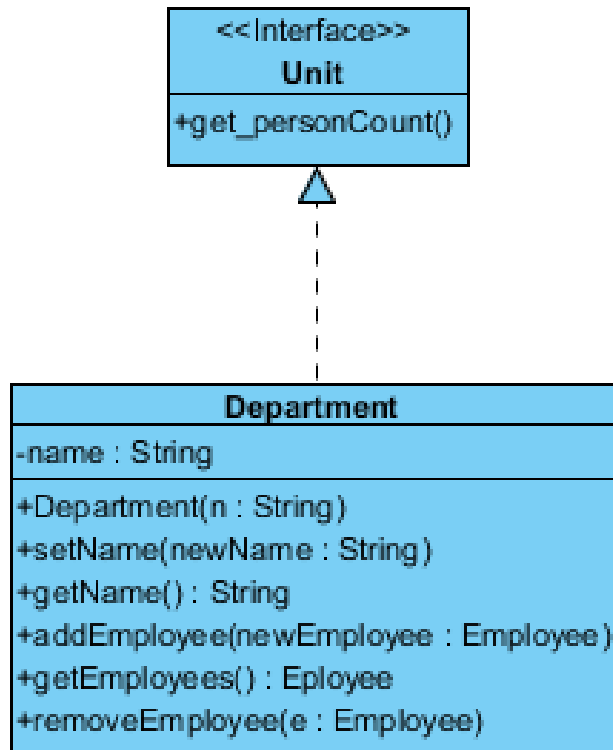
```
public class Man{
protected String name;
    protected String surname;
    public void setName(String newName){
        name = newName;
    }
    public String getName(){
        return name;
    }
    public void setSurname(String newSurname){
        name = newSurname;
    }
    public String getSurname(){
        return surname;
    }
}
```

```
public class Employee extends Man{
    private String position;
    public Employee(String n, String s, String p){
        name = n;
        surname = s;
        position = p;
    }
    public void setPosition(String newProfession){
        position = newProfession;
    }
    public String getPosition(){
        return position;
    }
}
```

# Отношения на диаграммах

## Реализация

*Клиент* реализует поведение поставляемое *поставщиком*  
Поставщик, как правило, является абстрактным классом или интерфейсом



# Отношения на диаграммах

## Реализация

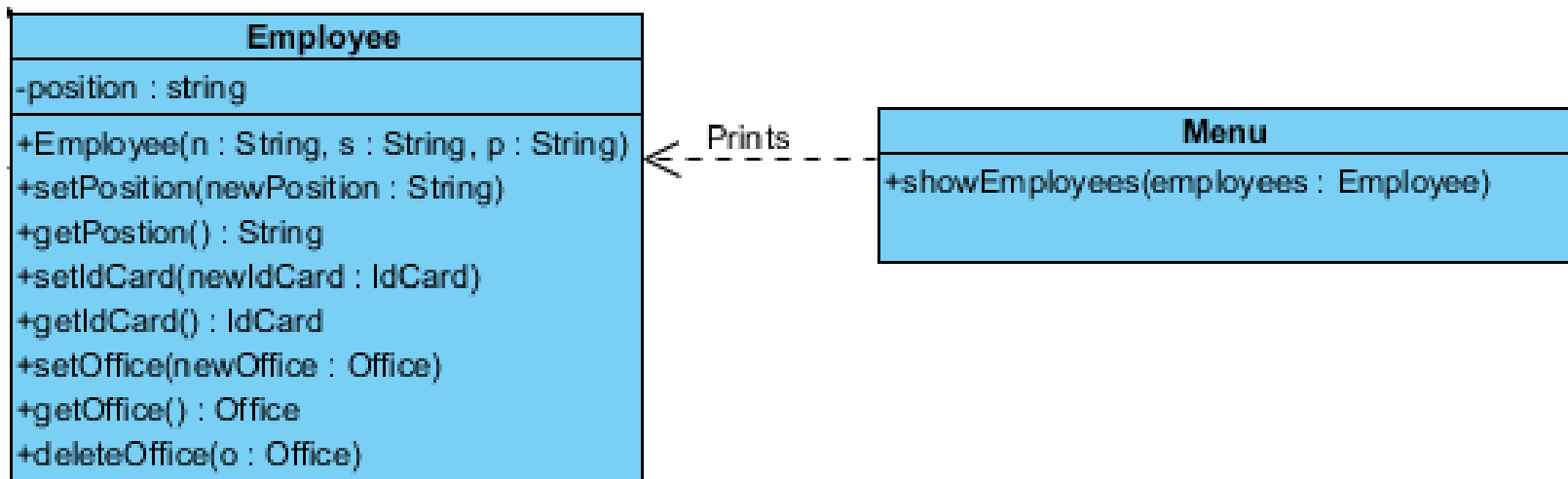
```
public interface Unit{  
    int getPersonCount();  
}
```

```
public class Department implements Unit{  
    ...  
    public int getPersonCount(){  
        return getEmployees().size();  
    }  
}
```

## Отношение зависимости

Указывает на возможность использования одним классом данных или операций другого класса (возможно, с некоторой модификацией)

Изменение спецификации класса-поставщика может повлиять на работу зависимого класса, но не наоборот



# Отношение зависимости

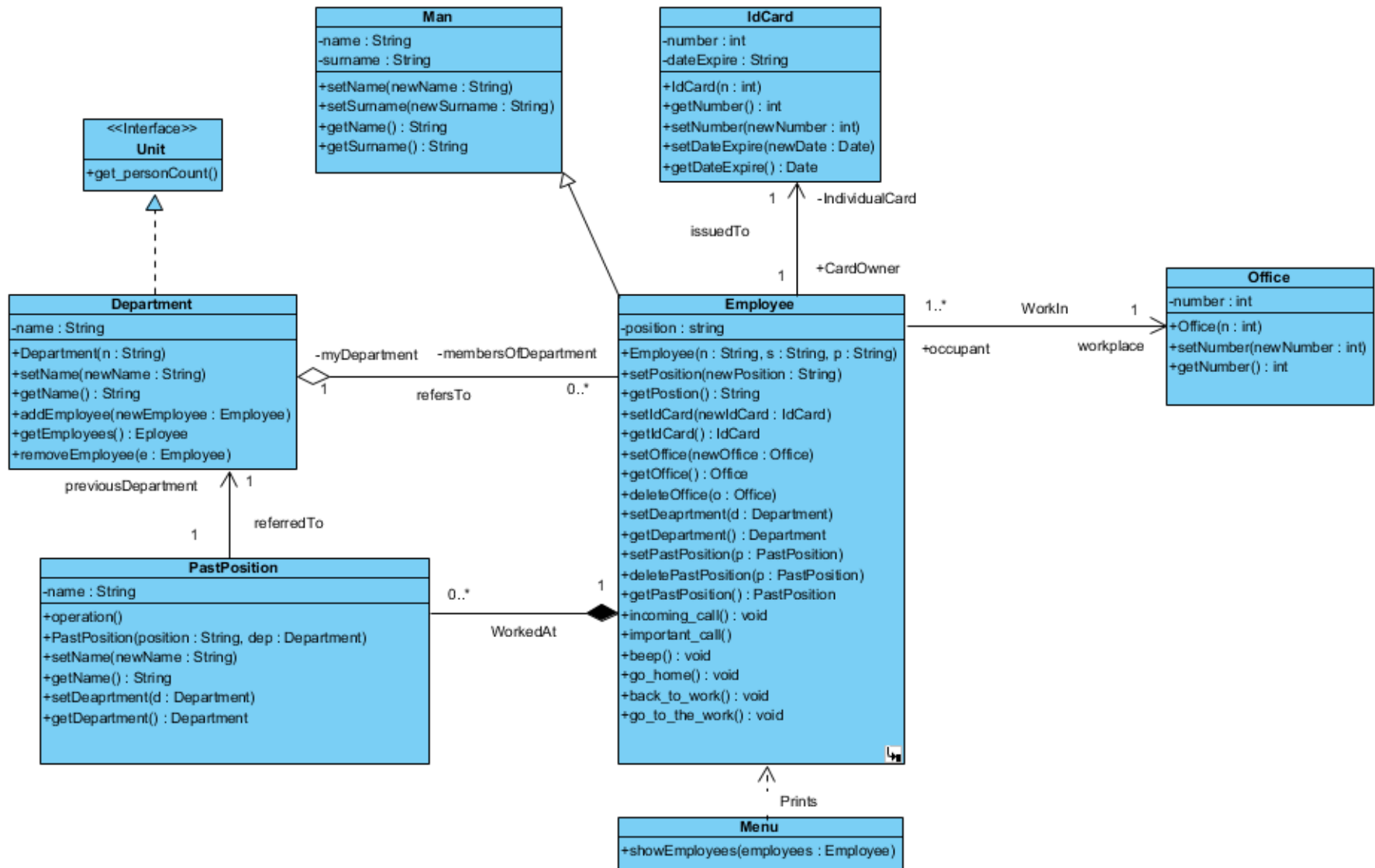
## Класс Menu

```
public class Menu{
    private static int i=0;
    public static void showEmployees(Employee[] employees){
        System.out.println("Список сотрудников:");
        for (i = 0; i < employees.length; i++){
            if(employees[i] instanceof Employee){
                System.out.println(employees[i].getName() + " - " + employees[i].getPosition());
            }
        }
    }
}
```

# Достоинства диаграмм классов

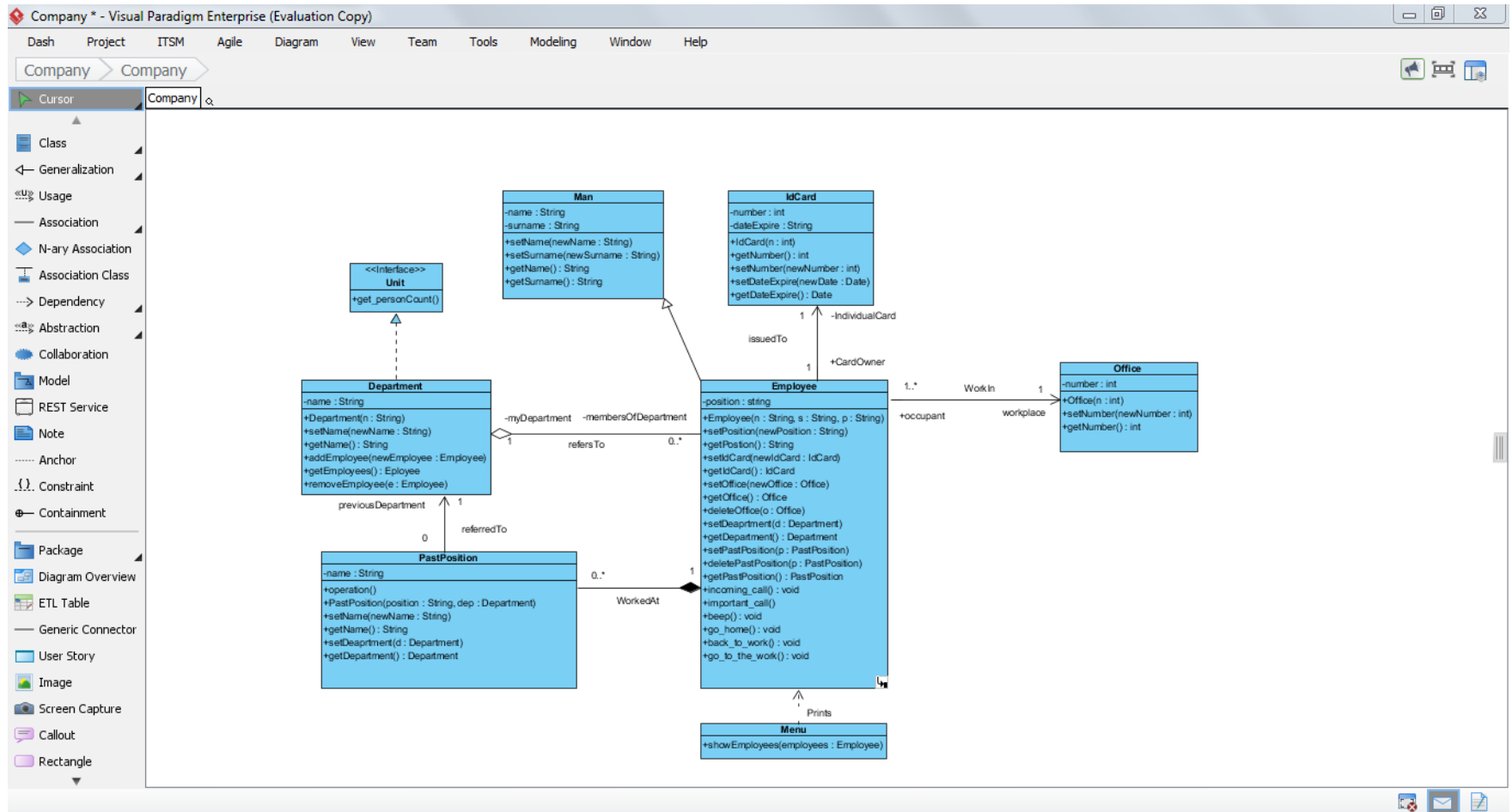
- Таким образом, диаграммы классов позволяют наглядно отображать отношения между классами, использующимися в реализации системы, отражая тем самым структурную схему системы
- Кодогенерация на основе диаграмм классов позволяет получить шаблон объявления классов с объявлением указанных атрибутов и методов

# Диаграмма классов системы управления кадрами





# Интерфейс Visual Paradigm



<https://www.visual-paradigm.com/download/>

Генерация кода по диаграмме классов: Tools -> Code -> Instant Generator

# Диаграммы состояний

Диаграммы состояний описывают поведенческие аспекты системы в терминах событий, переходов и состояний, в то время как диаграммы классов описывают структуру системы в терминах классов

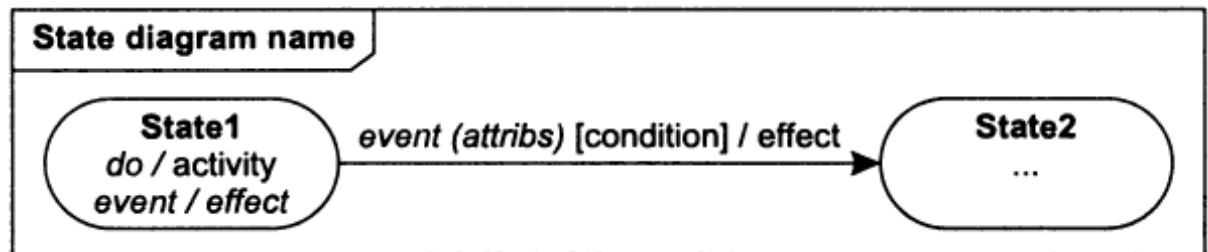
# Диаграмма состояний

Диаграмма состояний – направленный граф, вершинами которого являются состояния, а дугами – переходы

Диаграмма состояний прикрепляется к классу и служит описанием его функционирования (поведения)

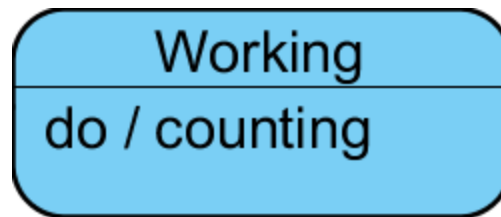
## Основные понятия

- Состояния
- События
- Переходы
- Действия



# Состояния

- Состояние – абстракция значений и связей объекта
- Ситуация, в которой объект
  - удовлетворяет некоторым условиям
  - выполняет какую-либо деятельность
  - ожидает некоторое событие
- Для определения состояния учитываются атрибуты класса, влияющие на его поведение



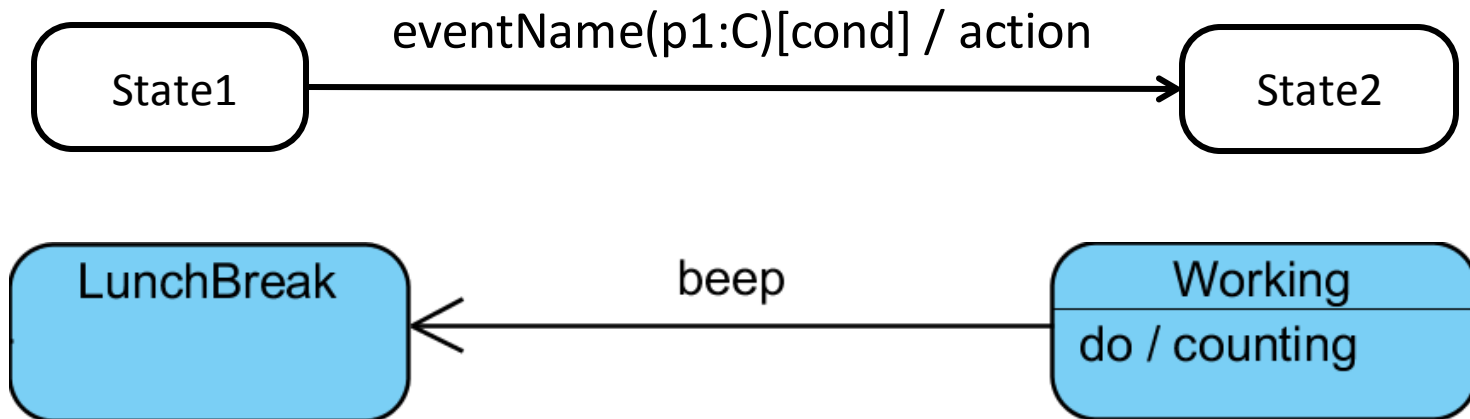
# События

Событие – происшествие, случившееся в определенный момент времени (мгновенное)

- Событие вызова
  - произошел вызов метода
- Событие изменения
  - произошло выполнение некоторого логического условия
  - when (  $a > b$  )
- Событие времени
  - достигнут определенный момент времени в системе
  - when (time = 1м 30с)
  - after (50 сек)
- Событие сигнала

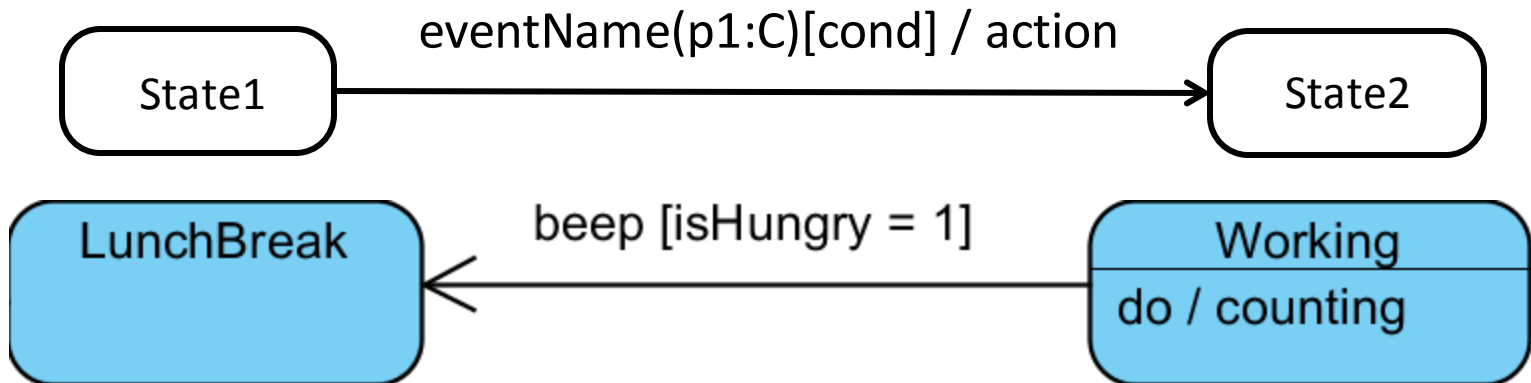
# Триггеры событий

- События в явном виде не представлены на диаграмме
- Переход вызывается триггером события
- Триггер события – событие, инициирующее переход из текущего состояния (при выполнении ограничений на переходе)
- Событие имеет параметры
- Триггер события имеет аргументы (фактические значения параметров события)



# Переходы

- События могут вызывать переходы
- Переход – мгновенная (в масштабе времени функционирования системы) смена состояния
- Переход характеризуется
  - событием
  - аргументами
  - ограничивающими условиями
  - действием

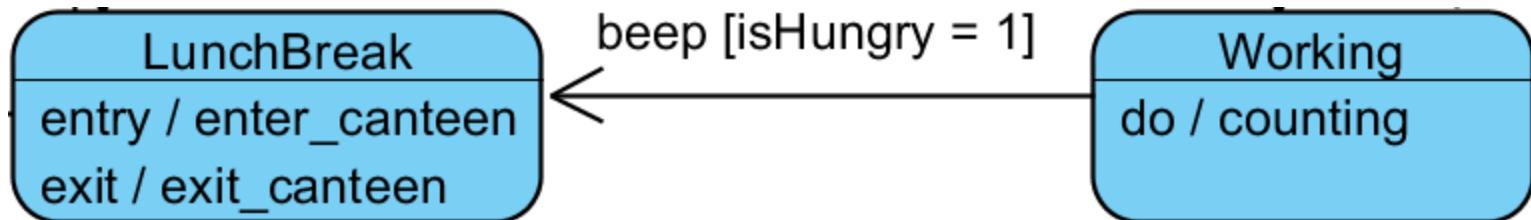


# Действия и деятельность

- Действие – атомарное непрерываемое поведение, вызываемое событием (как на переходе так и в состоянии)
- Деятельность – неатомарное поведение, реализуемое в состоянии
  - текущая деятельность (*do*)
  - деятельность на входе (*entry*)
  - деятельность на выходе (*exit*)



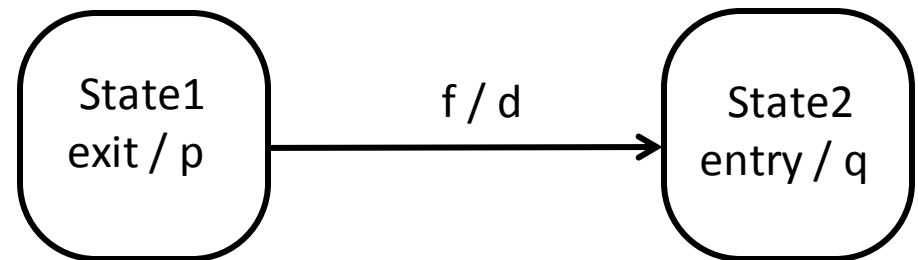
*entry*, *do*, *exit* –  
ключевые слова



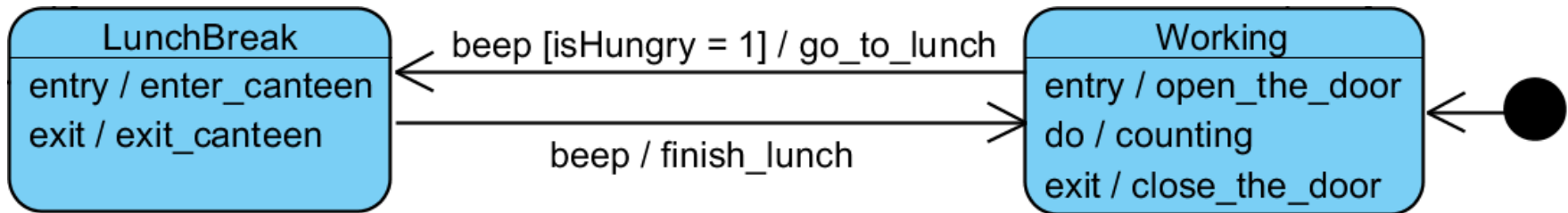


# Порядок выполнения действий при переходе

- Событие
- Выходное действие (*exit*)
- Действие на переходе (*action*)
- Входное действие (*entry*)
- Текущая деятельность (*do*)
- Событие

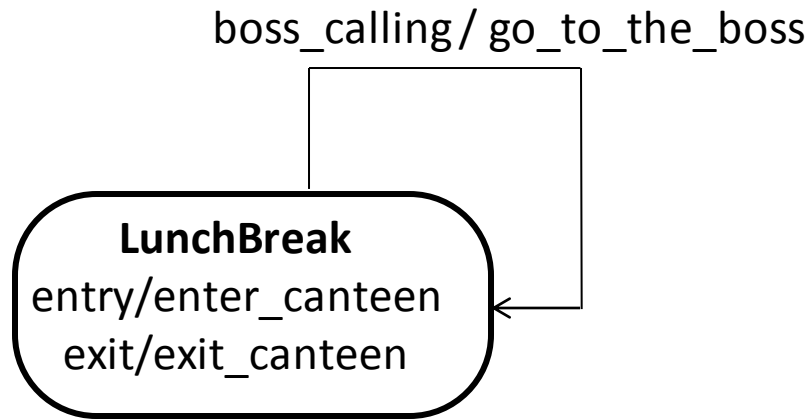


Последовательность действий : f, p, d, q



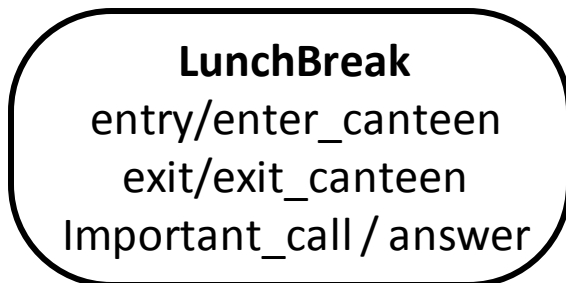
beep, close\_the\_door, go\_to\_lunch, enter\_canteen, beep, exit\_canteen, finish\_lunch, open\_the\_door, do

# Внутренние переходы и переходы по петле



Последовательность действий при наступлении события «boss\_calling»:

`boss_calling`  
`exit_canteen`  
`go_to_the_boss`  
`enter_canteen`

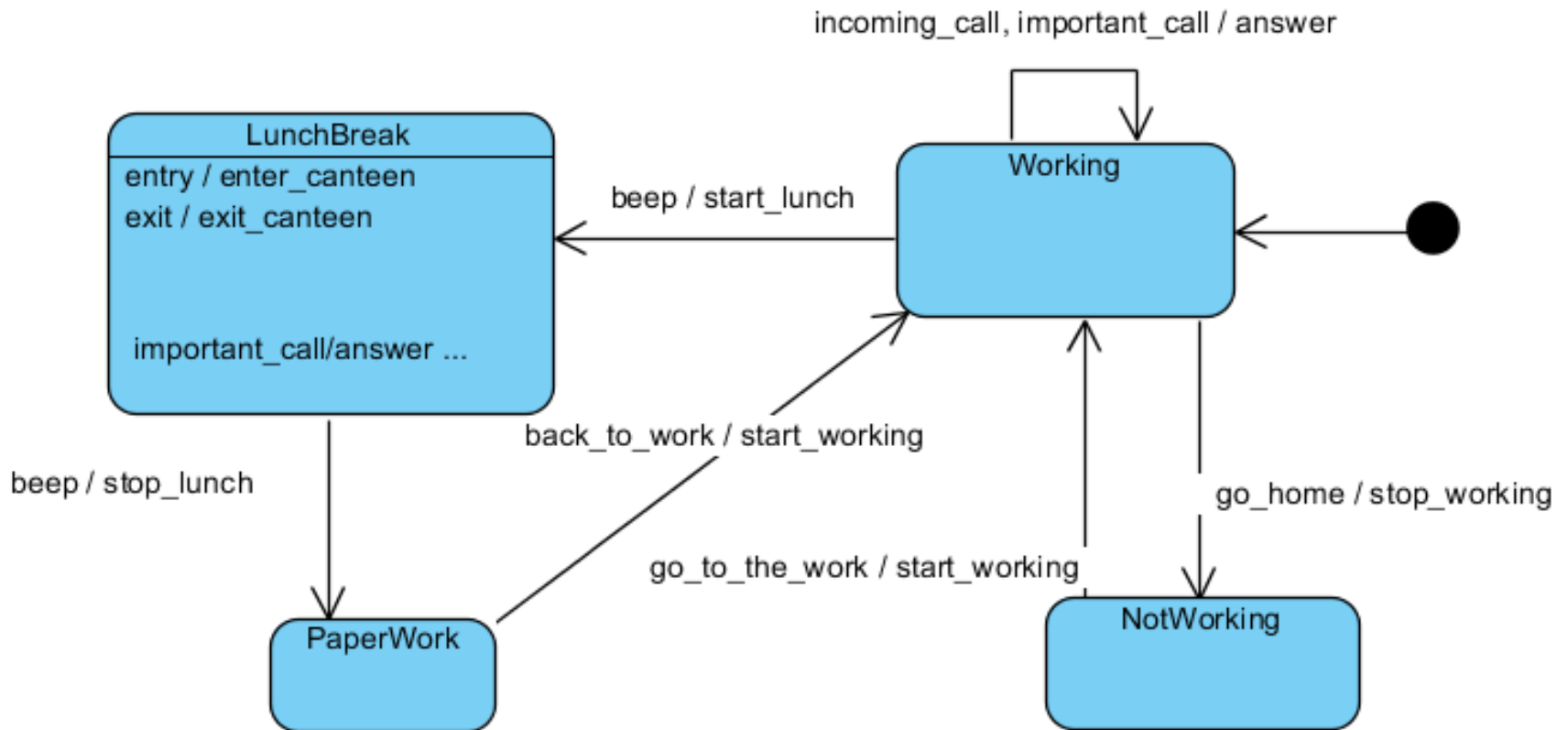


Последовательность действий при наступлении события «Important\_call»:

`Important_call`  
`answer`

# Пример

## Диаграмма состояний для класса Employee



# Генерация кода по диаграмме состояний

## Employee.java

```
package Company;

public class Employee {
    private EmployeeContext _fsm;
    private String position;

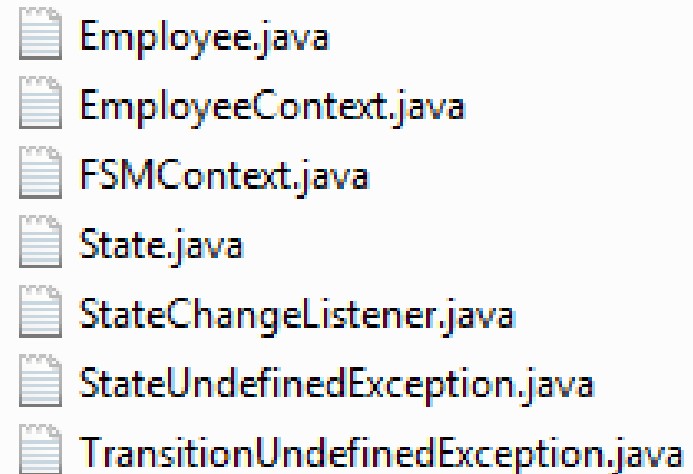
    public Employee() {
        _fsm = new EmployeeContext(this);
    }

    public EmployeeContext getContext() {
        return _fsm;
    }

    public Employee(String n, String s, String p) {
        throw new RuntimeException("Not Implemented!");
    }

    public void incoming_call() {
        _fsm.incoming_call();
    }


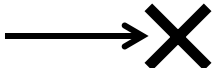
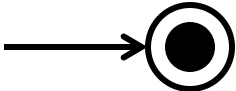
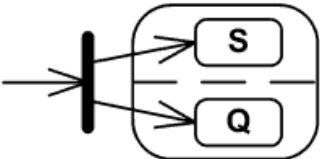
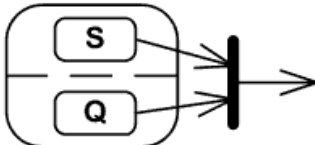
    public void important_call() {
        _fsm.important_call();
    }
}
```



- Employee.java
- EmployeeContext.java
- FSMContext.java
- State.java
- StateChangeListener.java
- StateUndefinedException.java
- TransitionUndefinedException.java

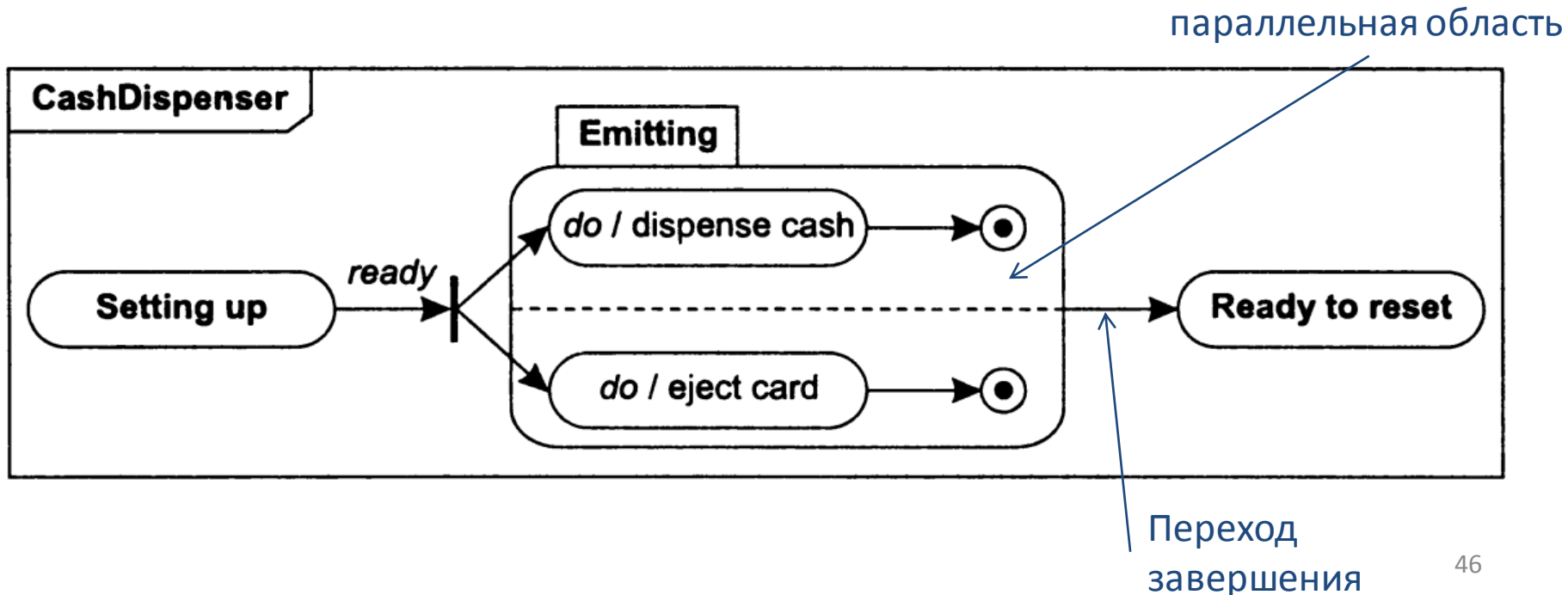
# Виды состояний

- Простое состояние
- Псевдосостояние – состояние с «особым» значением (вспомогательным)

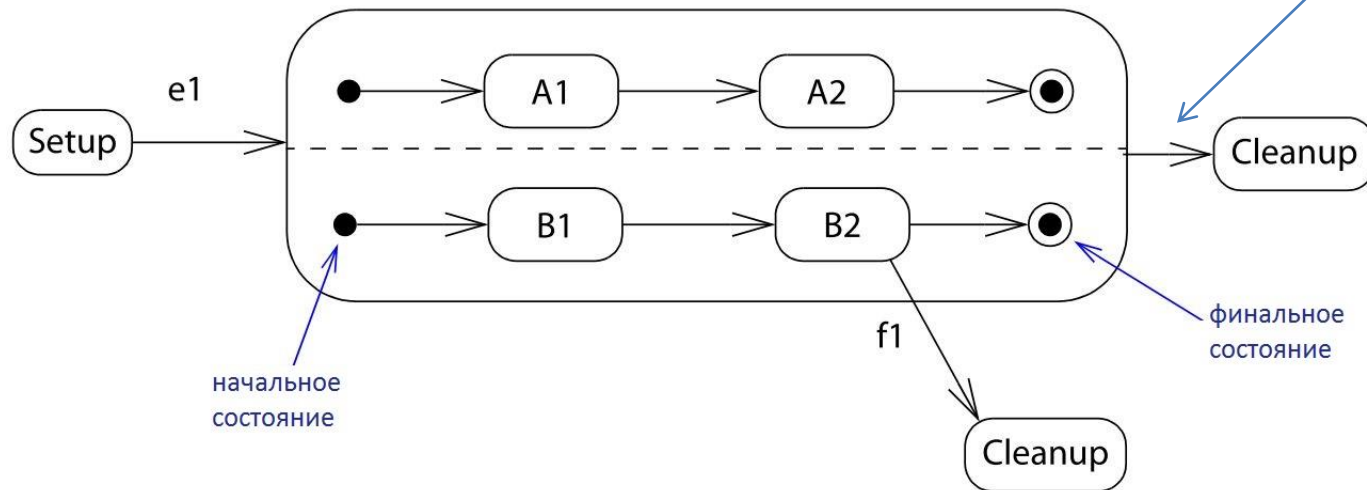
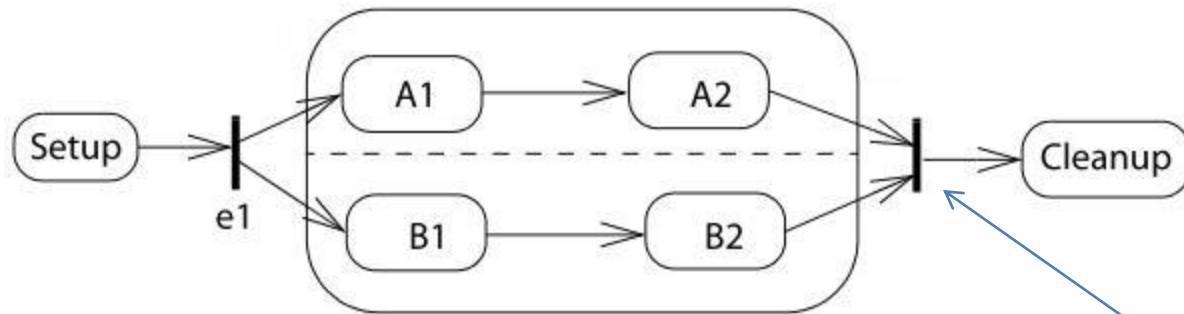
Начальное	Момент создания объекта	
Терминальное	Уничтожение объекта	
Финальное	Окончание фрагмента функционирования	
Разветвление	Переход в ортогональное композитное состояние	
Слияние	Переход из ортогонального композитного состояния	

# Композитные состояния

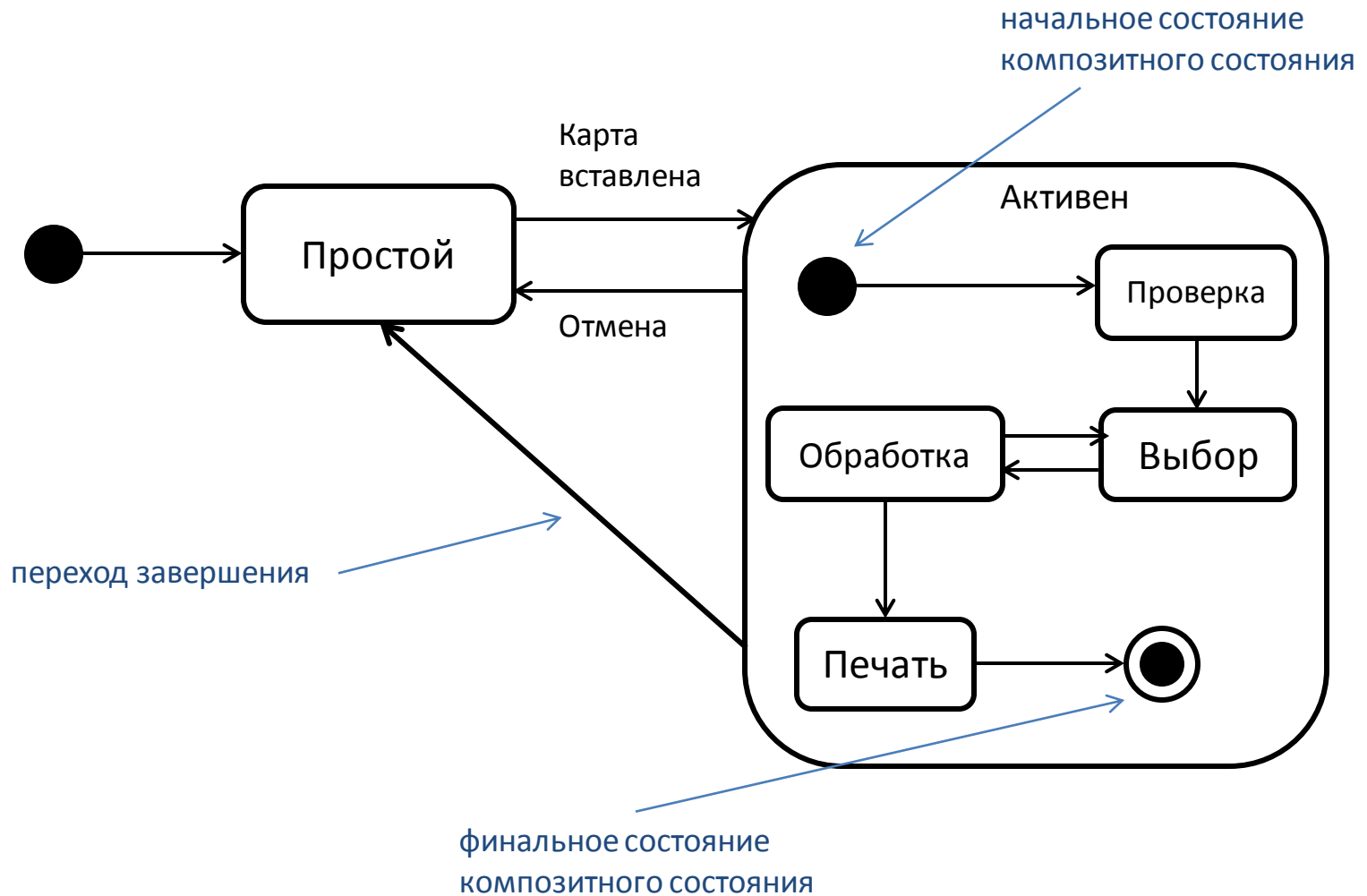
- Состояние, содержащее подсостояния, возможно разделенные на области
- Ортогональное состояние содержит две и более области, в каждой из которых в один момент времени может быть активным одно состояние



# Ортогональные состояния

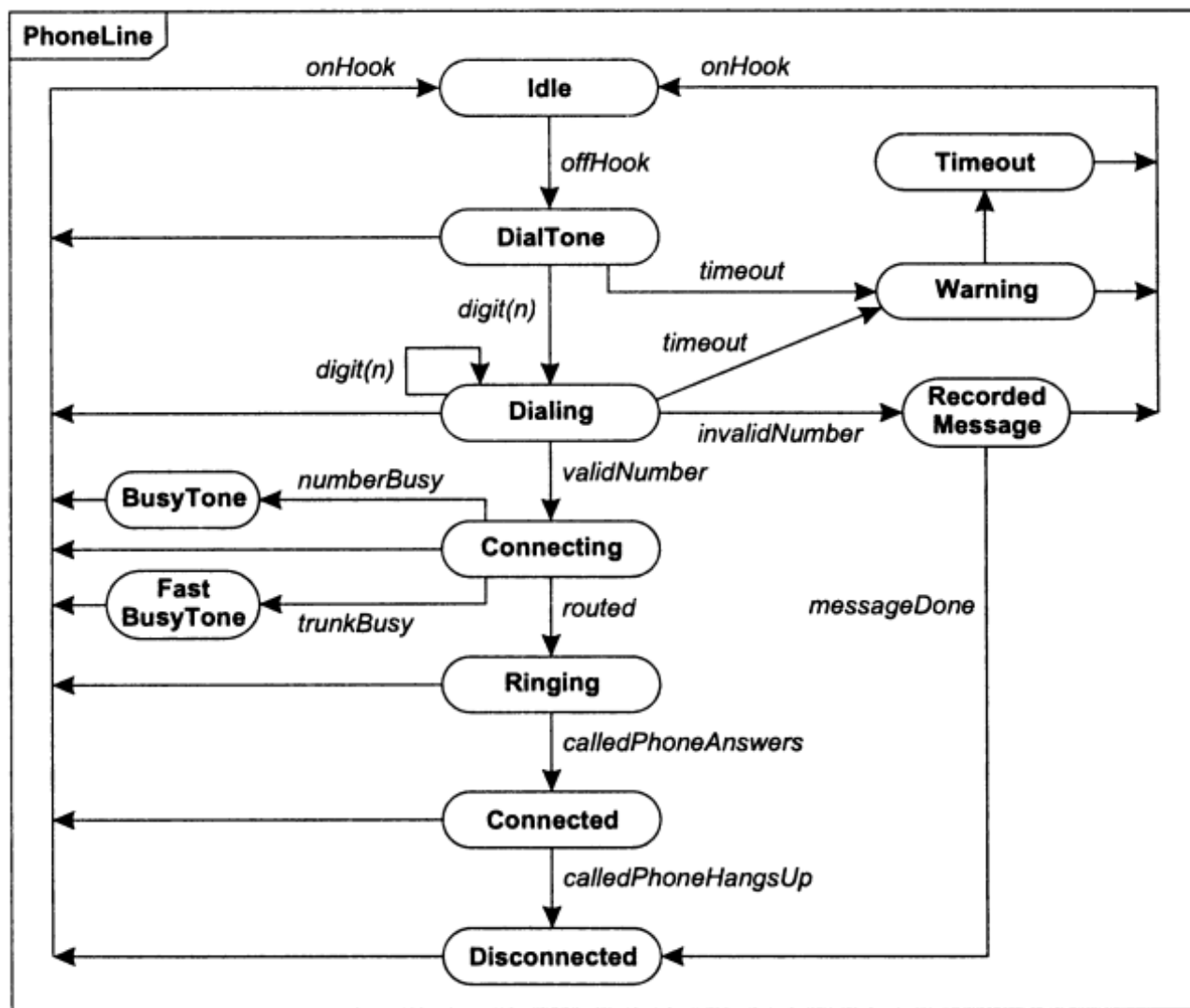


# Неортогональные состояния

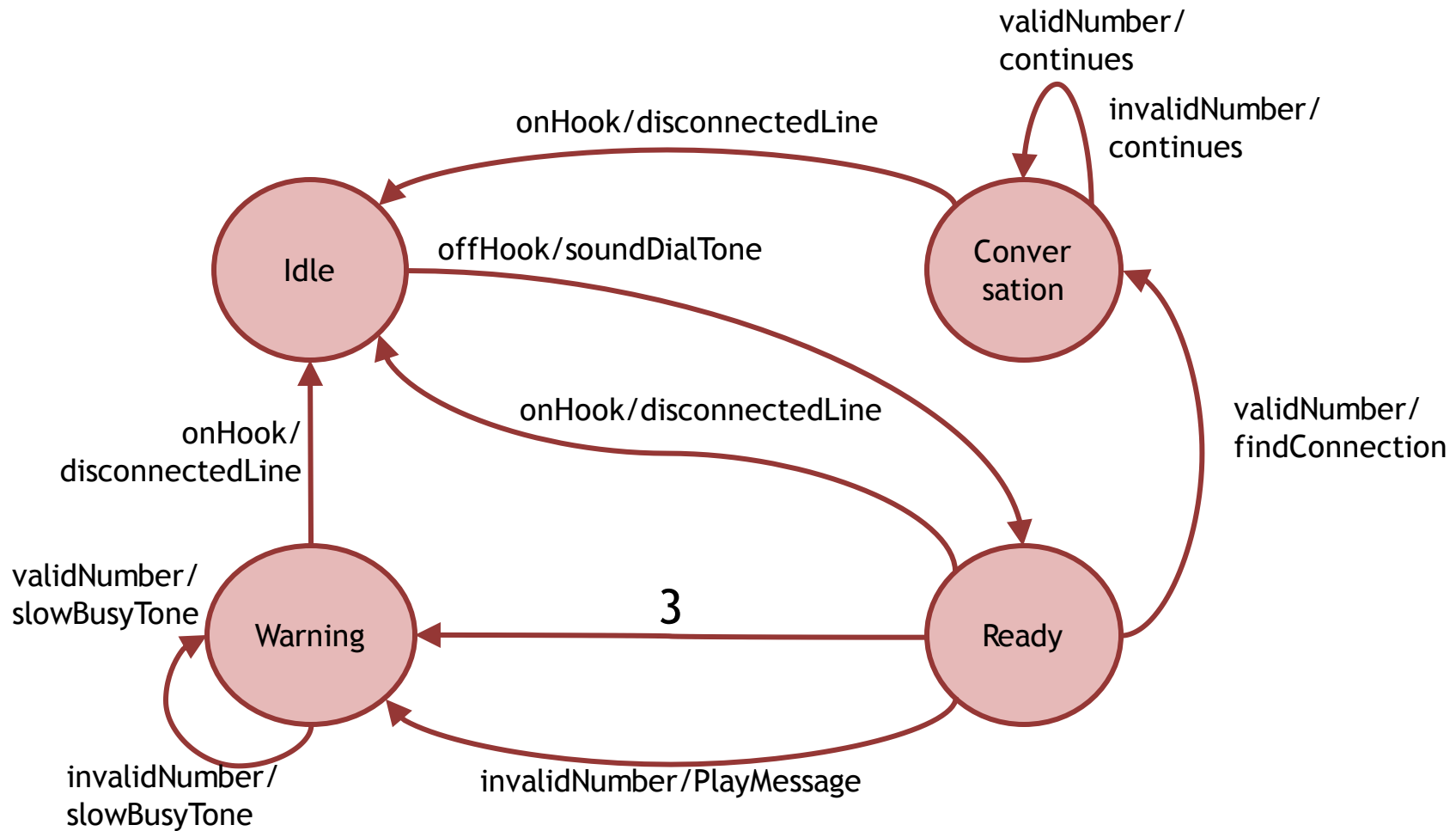




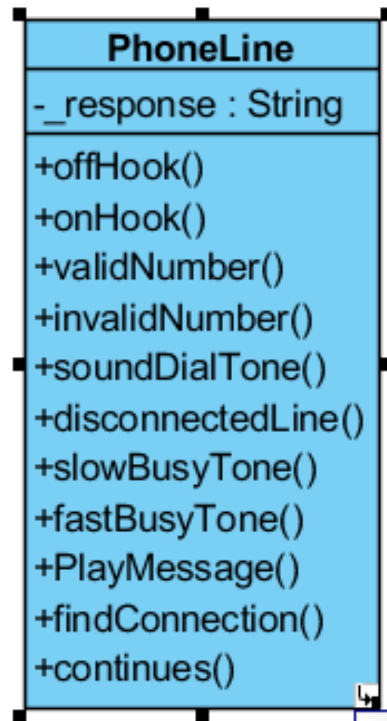
# Пример: «Телефонная линия»



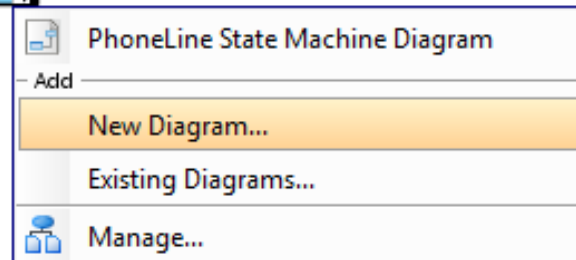
# Пример (автомат с таймаутами): телефонная линия



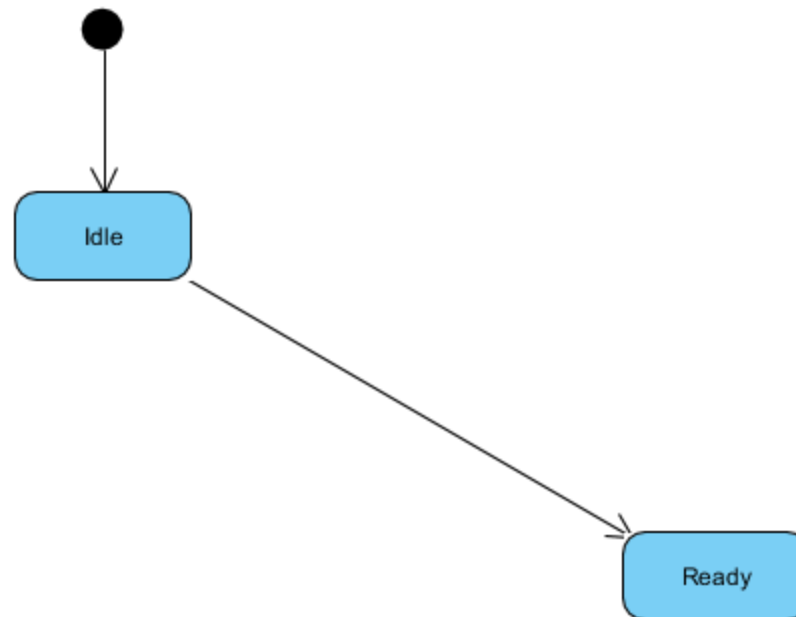
# Диаграмма классов системы «Телефонная линия»



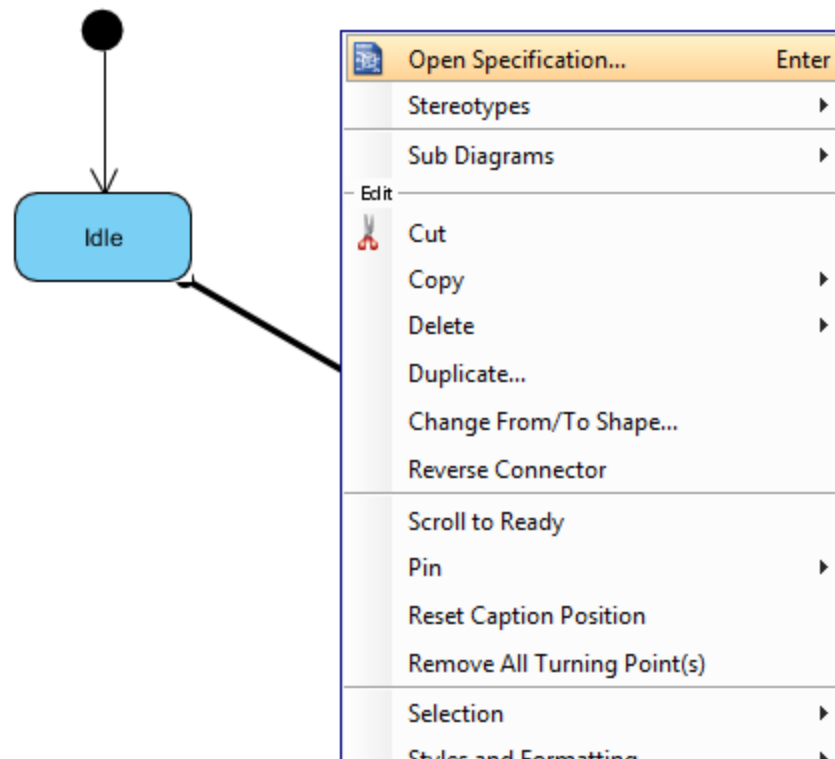
Для хранения реакции объекта на  
входное воздействие



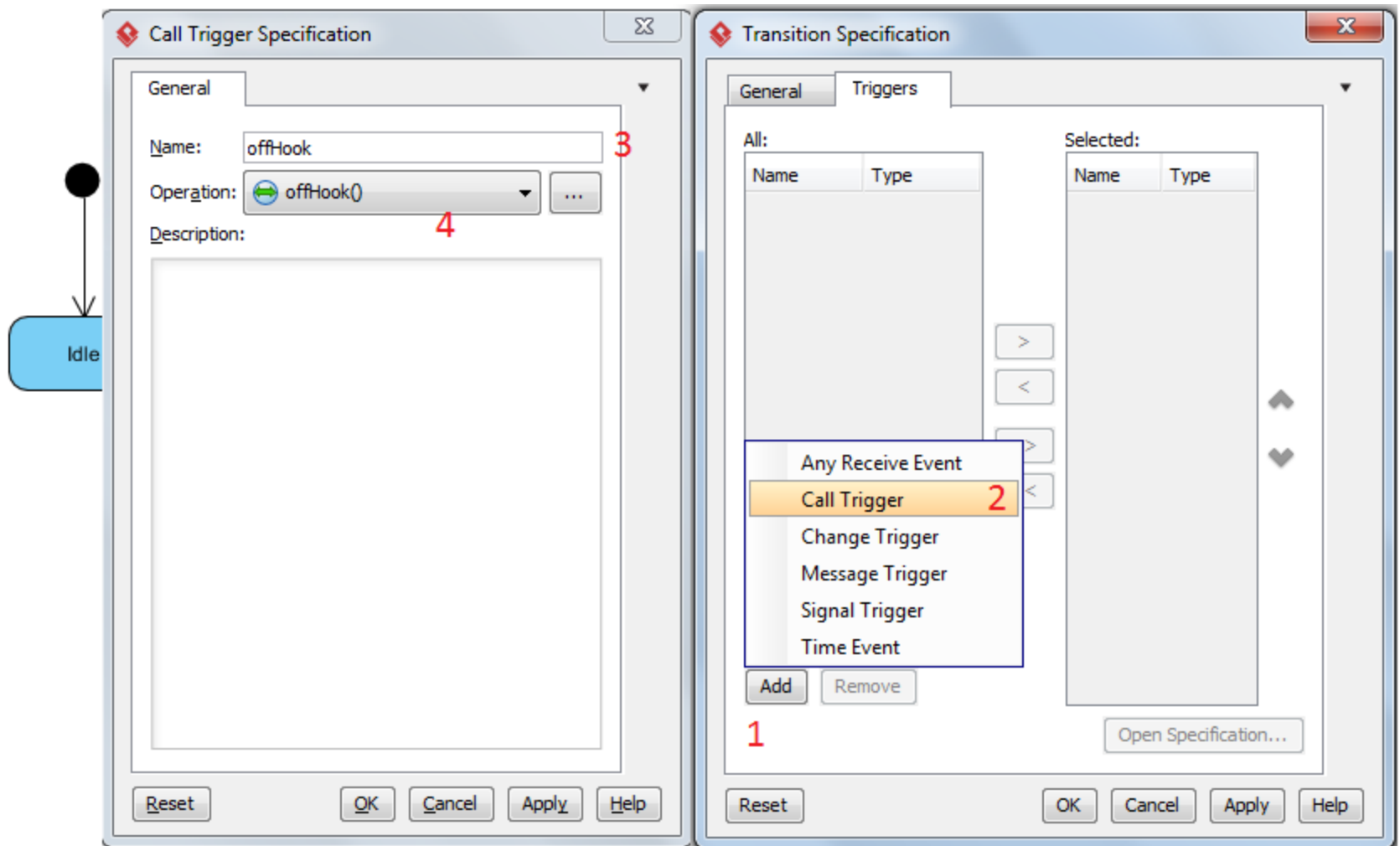
# Диаграмма состояний системы «Телефонная линия»



# Диаграмма состояний системы «Телефонная линия»



# Диаграмма состояний системы «Телефонная линия»



# Диаграмма состояний системы «Телефонная линия»

**Activity Specification**

☒ Create as child of Transition  
☐ Select a parent:  
☒ Show Model Only New Model

**PhoneLine**

**General** **Parameters**


Name:  **6**

Language:

Precondition:

Postcondition:

Body:

  **7**

Description:

☐ Single execution ☐ Read only ☐ Reentrant

Reset OK Cancel Apply Help

**Transition Specification**

**General** **Triggers**

Name:

Source:  ...

Target:  ...

Kind:

Effect:  ...

Redefined transition:  ...

Guard:  ...

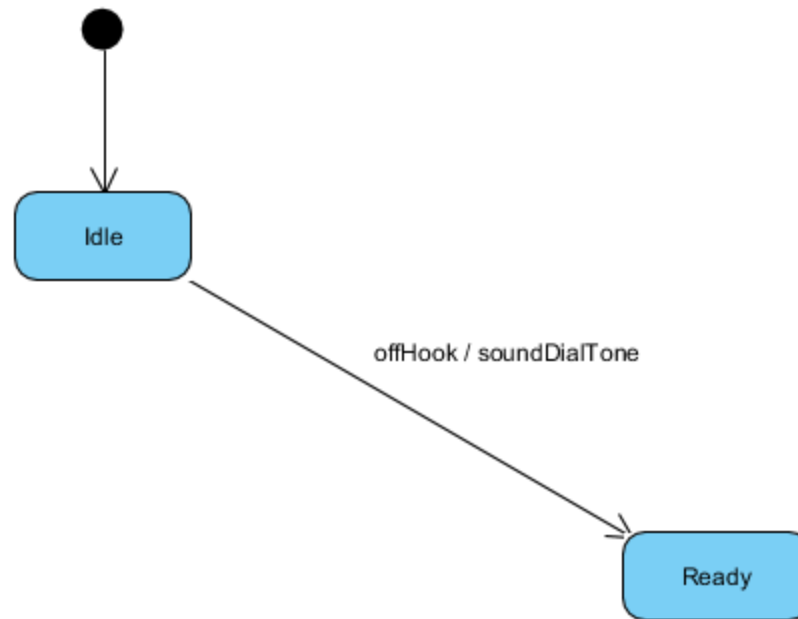
Select Constraint...

Description:

Reset OK Cancel Apply Help

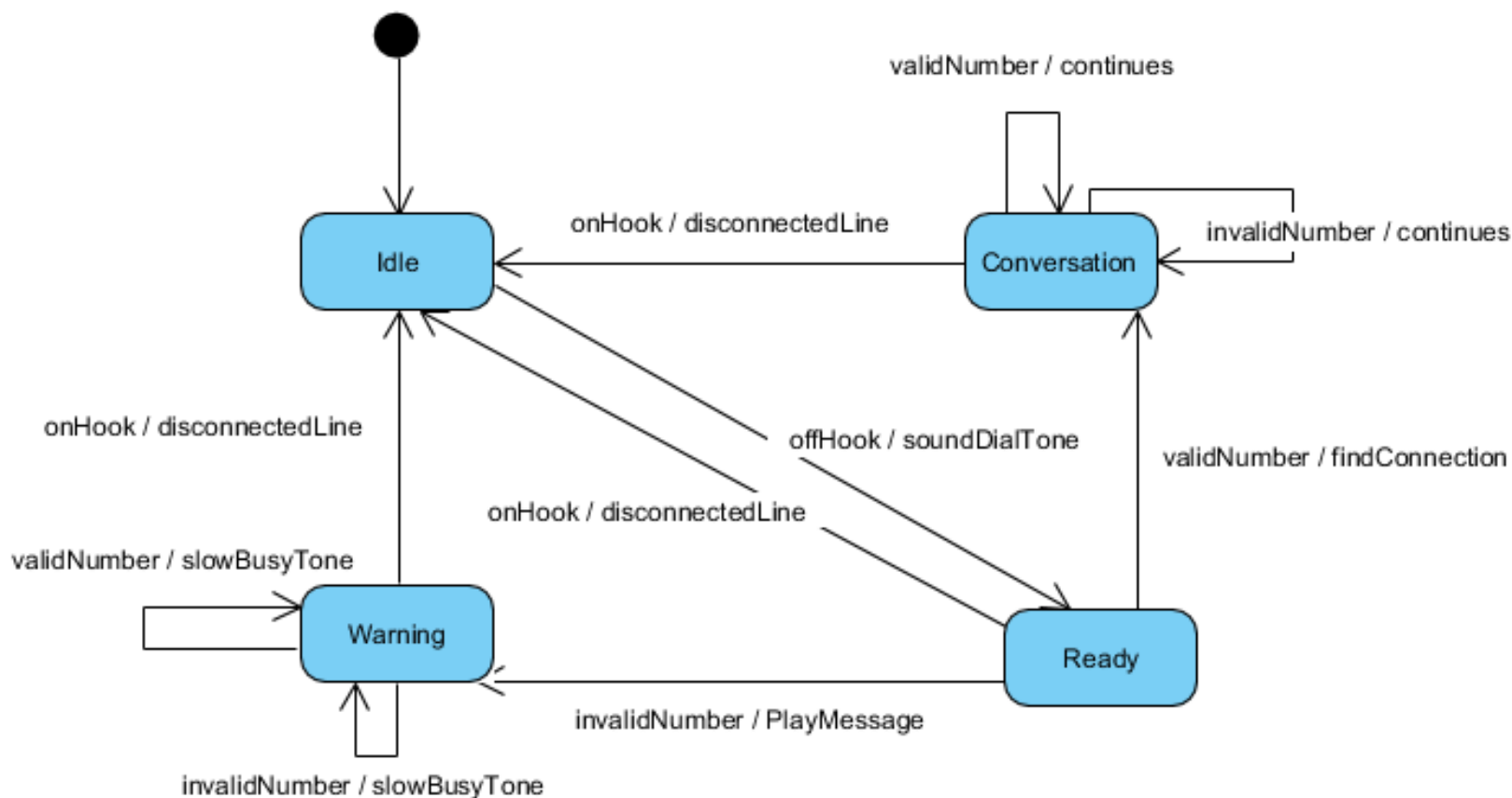
**5**

# Диаграмма состояний системы «Телефонная линия»



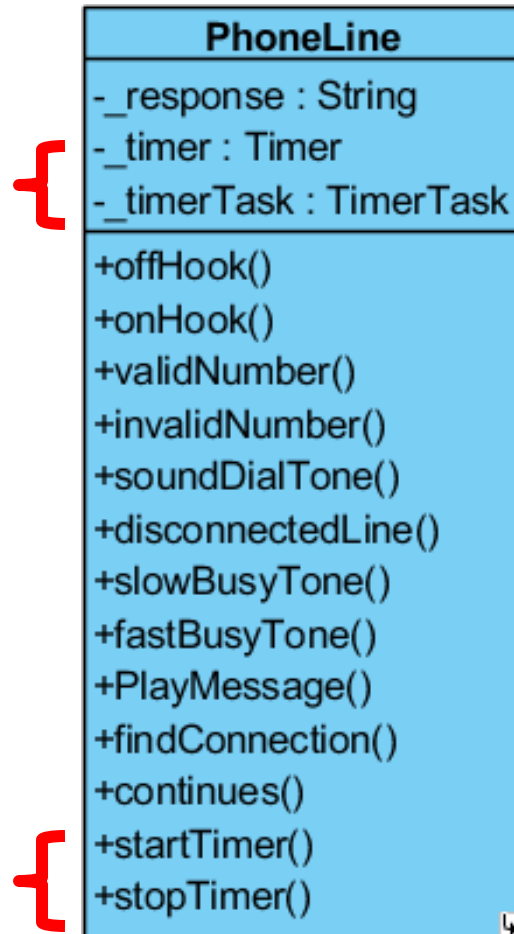


# Диаграмма состояний системы «Телефонная линия»



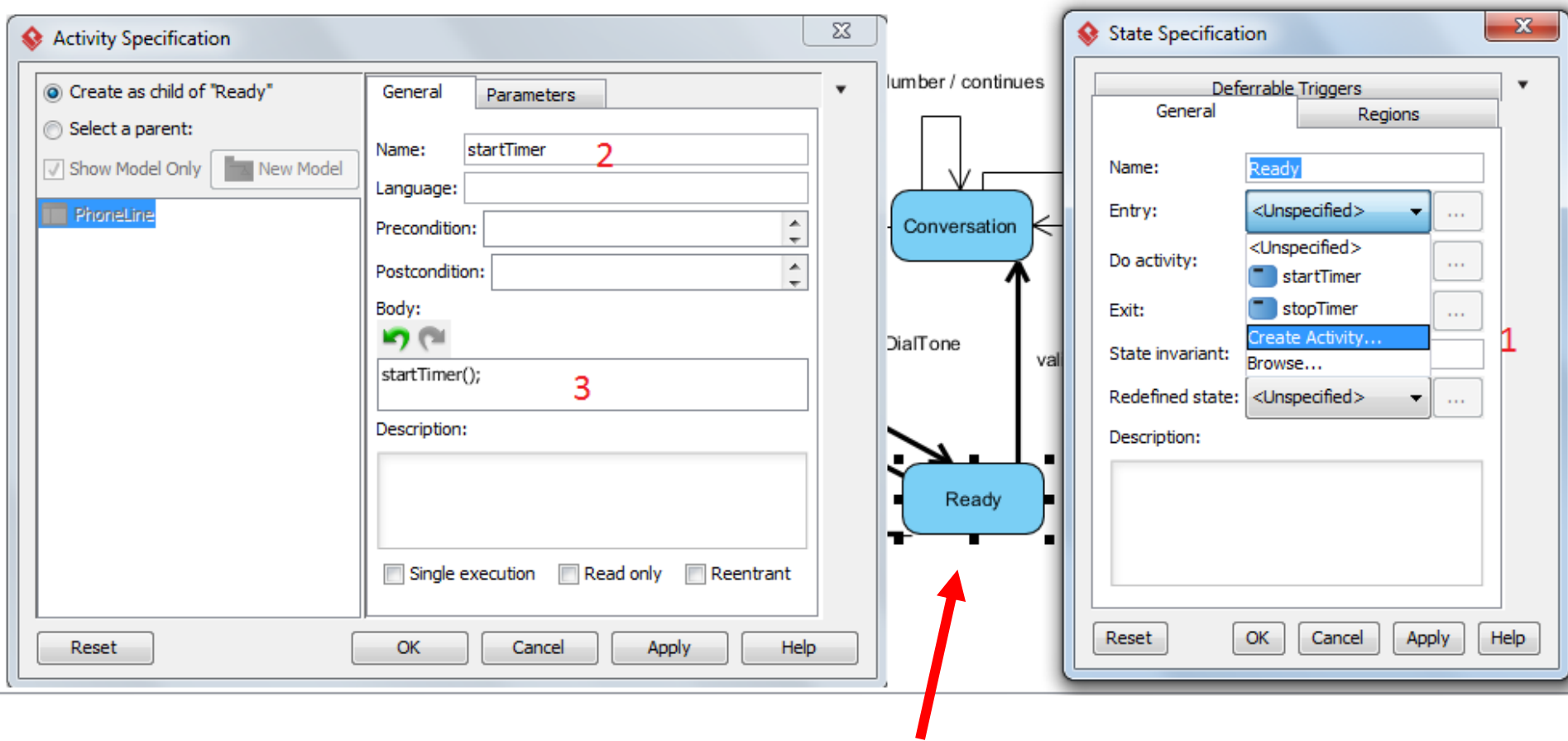
# Реализация таймаута в системе «Телефонная линия»

Диаграмма классов



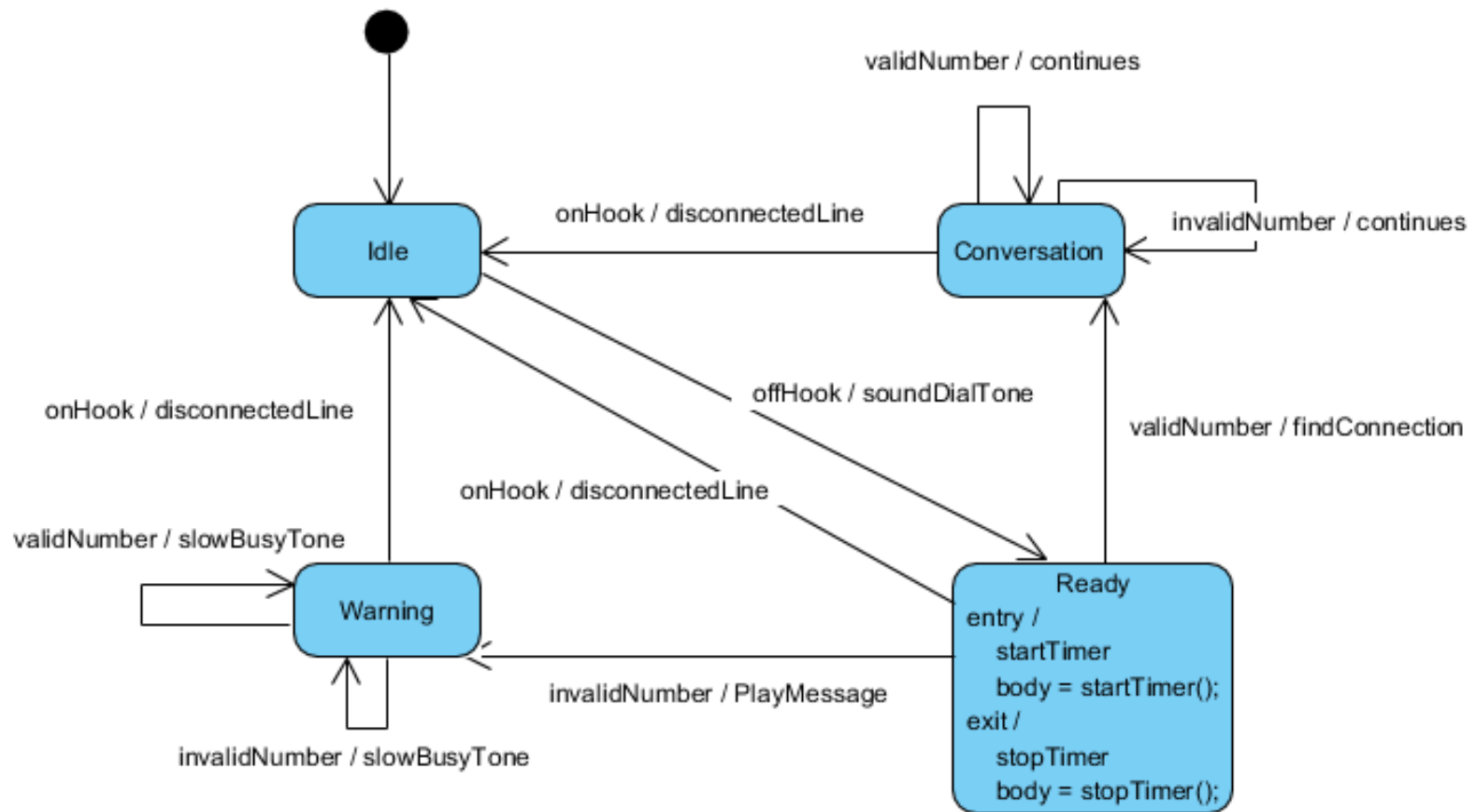
# Реализация таймаута в системе «Телефонная линия»

Диаграмма состояний



Состояние с таймаутом

# Реализация таймаута в системе «Телефонная линия»



# Генерация кода

Tools -> Code -> Generate State Machine Code

- FSMContext.java
- PhoneLine.java
- PhoneLineContext.java
- PhoneLineSample.java
- State.java
- StateChangeListener.java
- StateUndefinedException.java
- TransitionUndefinedException.java
- diagram.png
- PhoneLine.sm

## PhoneLine.java

```
public class PhoneLine {
    private PhoneLineContext _fsm;
    private String _response;
    private Timer _timer;
    private TimerTask _timerTask;

    public PhoneLine() {
        _fsm = new PhoneLineContext(this);
    }

    public PhoneLineContext getContext() {
        return _fsm;
    }

    public void offHook() {
        _fsm.offHook();
        throw new RuntimeException("Not Implemented!");
    }

    public void onHook() {
        _fsm.onHook();
        throw new RuntimeException("Not Implemented!");
    }
    .....
}
```

# Определение методов (один из вариантов)

## Методы - входы

```
public String offHook() {  
    _fsm.offHook();  
    return _response;  
}  
public String onHook() {  
    _fsm.onHook();  
    return _response;  
}  
.....  
.....
```

## Методы - выходы

```
public void soundDialTone() {  
    _response = "soundDialTone";  
}  
public void disconnectedLine() {  
    _response = "disconnectedLine";  
}  
.....  
.....
```

# Реализация таймаутов

Файл PhoneLine.java

```
import java.util.Timer;
import java.util.TimerTask;
public class PhoneLine {
    private PhoneLineContext _fsm;
    private String _response;
    private Timer _timer;
    private TimerTask _timerTask;

    .....
    public void startTimer ()
    {
        _timer = new Timer();
        _timerTask = new TimerTask() {
            @Override
            public void run() {
                (_fsm.getState()).Exit(_fsm);
                _fsm.clearState();
                _fsm.setState (PhoneLineContext.PhoneLineFSM.Warning); //указывается состояние в
                //которое система переходит по таймауту (Warning)
                (_fsm.getState()).Entry(_fsm); }
        };
        _timer.schedule(_timerTask, 300); //величина таймаута (1 такт = 100мс)
    }
    public void stopTimer()
    {
        _timerTask.cancel();
        _timerTask = null;
        _timer.cancel();
        _timer = null;
    }
}
```

# JUnit тесты

```
import static org.junit.jupiter.api.Assertions.*;
```

```
import org.junit.jupiter.api.*;
```

```
class PhoneLineTest {  
    PhoneLine p;
```

```
    @BeforeEach
```

```
    void setUp() throws Exception {  
        p = new PhoneLine();  
    }
```

```
    @AfterEach
```

```
    void tearDown() throws Exception {  
        p = null;  
    }
```

```
    @Test
```

```
    public void TestCase1() {  
        assertEquals(p.offHook(), "soundDialTone");  
        try {Thread.sleep(101);} catch (Exception ex) {}  
        assertEquals(p.onHook(), "disconnectedLine");  
    }
```

```
    @Test
```

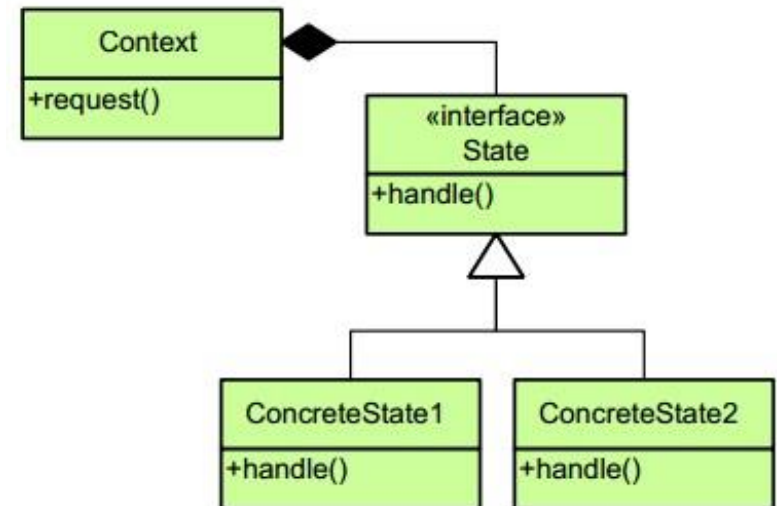
```
    public void TestCase2() {  
        .....  
    }  
}
```



# Паттерн «Состояние»

## PhoneLine.java

```
public class PhoneLine {  
    private PhoneLineContext _fsm;  
    private String _response;  
    private Timer _timer;  
    private TimerTask _timerTask;  
    public PhoneLine() {  
        _fsm = new PhoneLineContext(this);  
    }  
  
    public PhoneLineContext getContext() {  
        return _fsm;  
    }  
  
    public void offHook() {  
        _fsm.offHook();  
        throw new RuntimeException("Not Implemented!");  
    }  
  
    public void onHook() {  
        _fsm.onHook();  
        throw new RuntimeException("Not Implemented!");  
    }  
    .....  
    .....  
}
```



# Формат .sm

```
%start PhoneLineFSM::Idle
%class PhoneLine
%package phl

%map PhoneLineFSM
%%
Ready
Entry {startTimer();}
Exit {stopTimer();}
{
    onHook    Idle {disconnectedLine();}
    invalidNumber Warning {PlayMessage();}
    validNumber Conversation {findConnection();}
}

Warning
{
    validNumber    Warning {slowBusyTone();}
    onHook    Idle {disconnectedLine();}
    invalidNumber Warning {slowBusyTone();}
}

Conversation
{
    validNumber    Conversation {continues();}
    invalidNumber    Conversation {continues();}
    onHook    Idle {disconnectedLine();}
}

.....
%%
```

<Состояние>

```
{
    <Действие> <Состояние> { <Реакция> }
}
```

<Состояние>

Entry

```
{
    <Действие>
}
```

← Выполняется каждый раз при входе в состояние

Exit

```
{
    <Действие>
}
```

← Выполняется каждый раз при выходе из состояния

```
{
    <Действие> <Состояние> { <Реакция> }
}
```

State Machine Compiler (SMC)

<http://smc.sourceforge.net/>

# FSMTest

## Формат (.fsm) описания автоматов

```
F 2          //2 - временной детерминированный автомат
s 2          // число состояний
i 2          // число входных символов
o 2          // число выходных символов
n0 0         // начальное состояние
p 4          // число переходов
0 0 1 0 0    //presState input nextState output Delay
0 1 0 1 0    // <Состояние> <Действие> <Состояние> <Реакция> <Задержка>
1 0 1 1 0
1 1 0 0 0
0 2 1        // presState timeout nextState
1 -1 1       // таймаут отсутствует
```

# Конечно автоматная абстракция

F 2  
s 2  
i 2  
o 2  
n 0 0  
p 4  
0 0 1 0 0  
0 1 0 1 0  
1 0 1 1 0  
1 1 0 0 0  
0 2 1  
1 -1 1

tfsm-fsm.exe

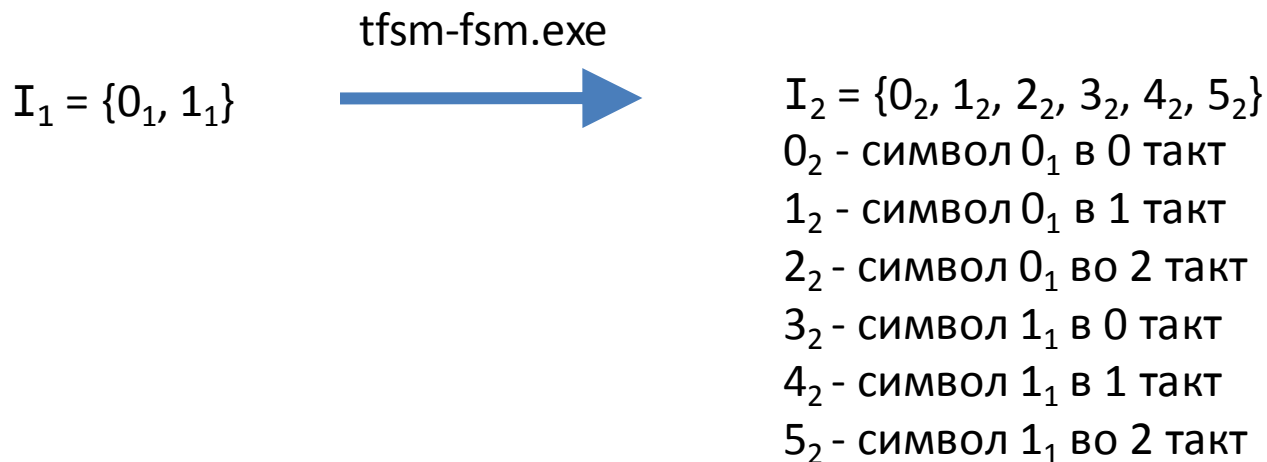


F 0      // детерминированный автомат  
s 2  
i 6  
o 2  
n 0 0  
p 12  
0 0 1 0  
0 1 1 0  
0 2 1 1  
0 3 0 1  
0 4 0 1  
0 5 0 0  
1 0 1 1  
1 1 1 1  
1 2 1 1  
1 3 0 0  
1 4 0 0  
1 5 0 0

# Конечно автоматная абстракция

Каждый временной входной символ преобразуется в новый (не временной) входной символ

Например, если значение наибольшего таймаута равно 2, то множество входных символов преобразуется следующим образом



# Тестирование

1. Перевести PhoneLine.sm в PhoneLine.fsm
2. Добавить в PhoneLine.fsm таймауты
3. Построить конечно автоматную абстракцию  
`$tfsm-fsm.exe PhoneLine.fsm PhoneLine_abs.fsm`
4. Построить тест методом обхода графа переходов  
`$transitionTour.exe PhoneLine_abs.fsm PhoneLine_abs_test.txt`
5. Написать JUnit тесты на основе полученного конечно-автоматного теста

-----

Visual Paradigm (<https://www.visual-paradigm.com/download/>)

# Тестирование

Например

Входные символы захешированы так:

offHook – 0

onHook – 1

И т.д.

Выходные символы захешированы так:

soundDialTone – 0

disconnectedLine – 1

И т.д.

Максимальный таймаут равен 3

После построения теста по конечно автоматной абстракции тестовая последовательность 2/0 5/1 соответствует временной тестовой последовательности (0,2)/0 (1,1)/1

```
public void TestCase1() {  
    try {Thread.sleep(200); } catch (Exception ex) {}  
    assertEquals(p.offHook(), "soundDialTone");  
    try {Thread.sleep(100); } catch (Exception ex) {}  
    assertEquals(p.onHook(), "disconnectedLine");  
}
```

Спасибо за внимание!

Лапутенко Андрей Владимирович  
E-mail: [laputenko.av@gmail.com](mailto:laputenko.av@gmail.com)