МИНОБРНАУКИ РОССИИ

Передовые инженерные школы

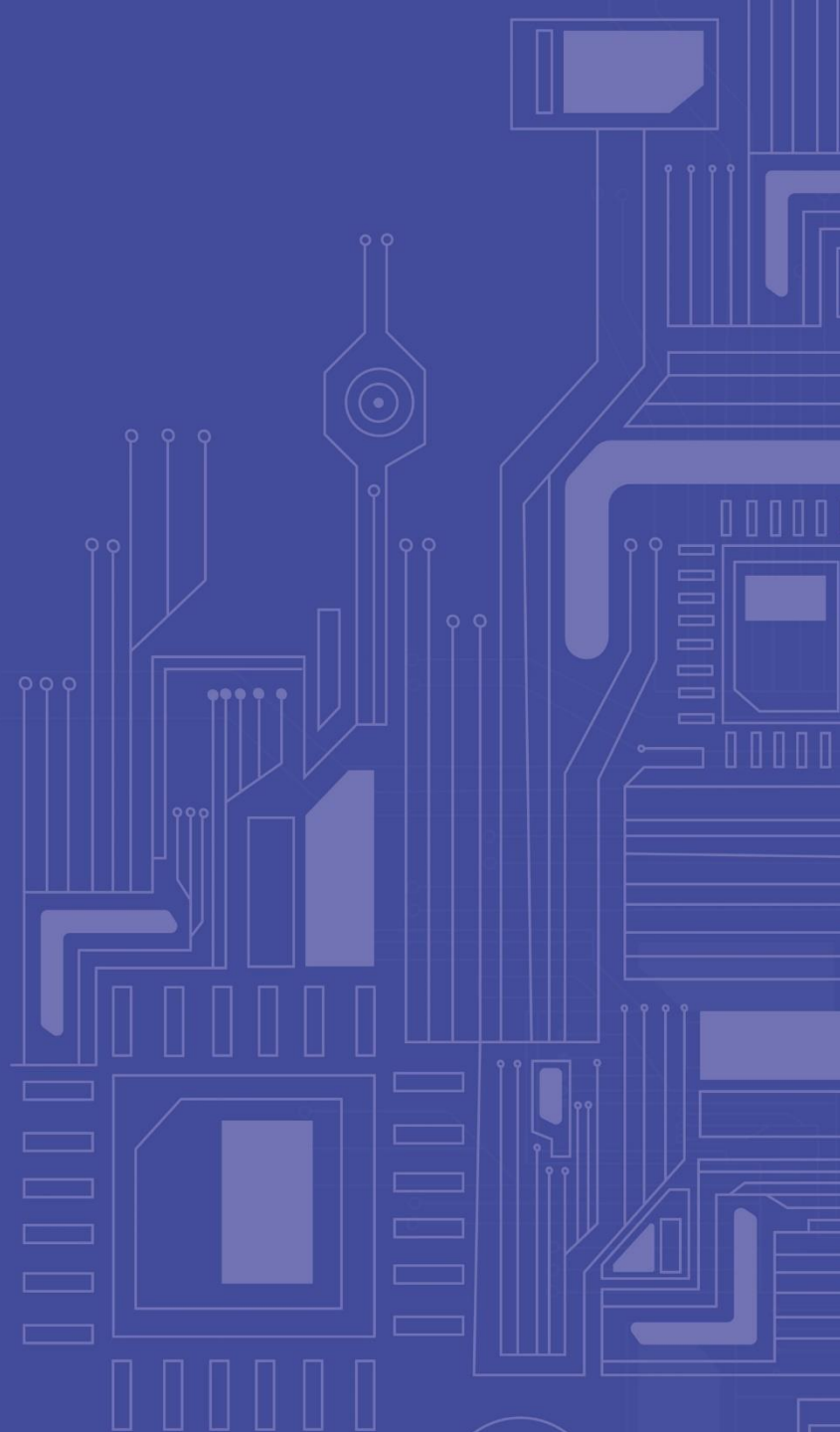
СИСТЕМЫ УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ

Лекция 5

ПЛАН ЛЕКЦИИ

- Более сложные элементы ER-модели
  - Получение реляционной схемы из ER-диаграммы
    - Язык UML
      - Язык OCL

БОЛЕЕ СЛОЖНЫЕ ЭЛЕМЕНТЫ ER-МОДЕЛИ



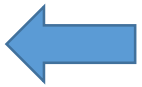
БОЛЕЕ СЛОЖНЫЕ ЭЛЕМЕНТЫ ER-МОДЕЛИ (1/2)

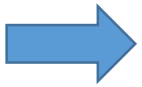
Подтипы и супертипы сущностей. Подобно тому как это делается в языках программирования с развитыми типовыми системами (например, в языках объектно-ориентированного программирования), в ER-модели поддерживается возможность определения нового типа сущности путем наследования некоторого супертипа сущности. Механизм наследования в ER-модели обладает несколькими особенностями: в частности, интересные нюансы связаны с необходимостью графического изображения этого механизма.

Уточняемые степени связи. Иногда бывает полезно определить возможное количество экземпляров сущности, участвующих в данной связи (например, ввести ограничение, связанное с тем, что служащему разрешается участвовать не более чем в трех проектах одновременно). Для выражения этого семантического ограничения разрешается указывать на конце связи ее максимально допустимую или обязательную степень.

БОЛЕЕ СЛОЖНЫЕ ЭЛЕМЕНТЫ ER-МОДЕЛИ (2/2)

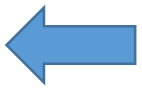
Взаимно исключающие связи. Для заданного типа сущности можно определить такой набор типов связи с другими типами сущности, что для каждого экземпляра заданного типа сущности может (если набор связей является необязательным) или должен (если набор связей обязателен) существовать экземпляр только одной связи из этого набора.





Каскадные удаления экземпляров сущностей. Некоторые связи бывают настолько сильными (конечно, в случае связи «один ко многим»), что при удалении опорного экземпляра сущности (соответствующего концу связи «один») нужно удалить и все экземпляры сущности, соответствующие концу связи «многие». Соответствующее требование каскадного удаления можно специфицировать при определении связи.

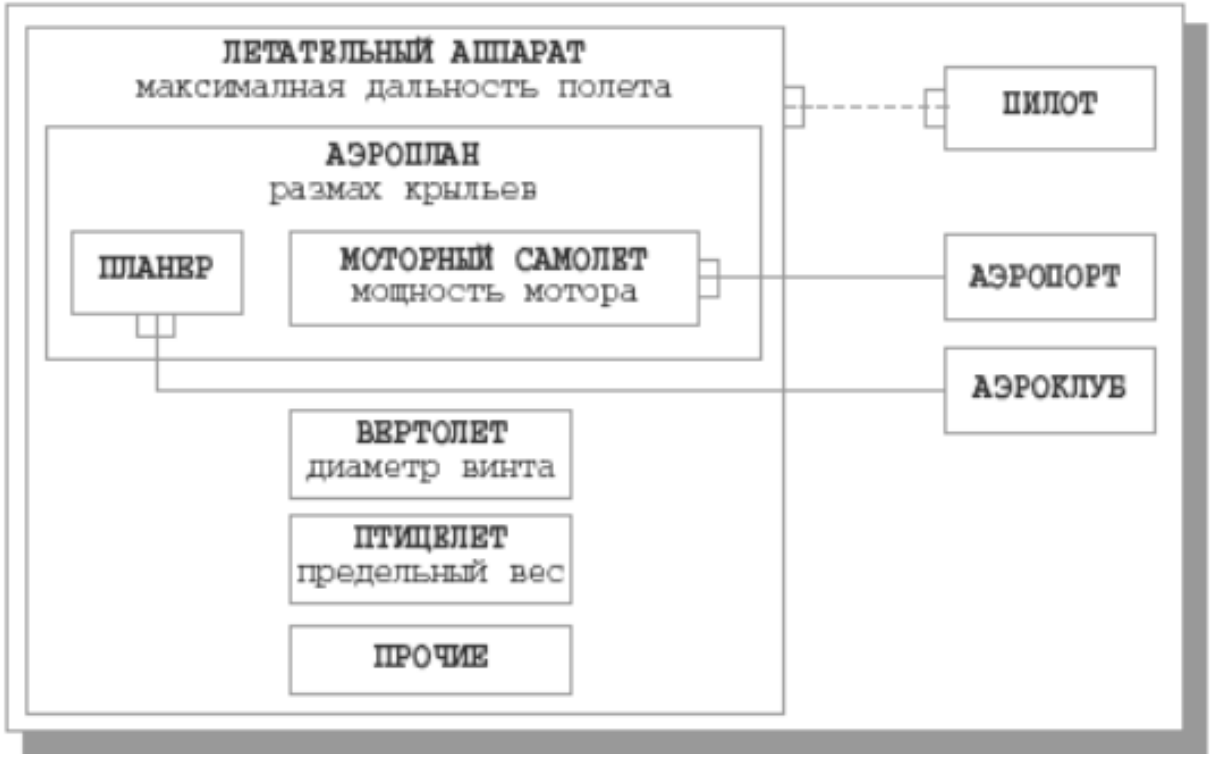
Домены. Как и в случае реляционной модели данных, в некоторых случаях полезна возможность определения потенциально допустимого множества значений атрибута сущности (домена).



НАСЛЕДОВАНИЕ ТИПОВ СУЩНОСТИ И ТИПОВ СВЯЗИ

Если у типа сущности A имеются подтипы $B_1, B_2,..., B_n$, то:

- **Включение**: любой экземпляр типа сущности B_1, $B_2,..., B_n$ является экземпляром типа сущности A .
- **Отсутствие собственных экземпляров у супертипа сущности:** если a является экземпляром типа сущности A, то a является экземпляром некоторого подтипа сущности B_i (i = 1, 2, ..., n).
- **Разъединенность подтипов:** ни для каких подтипов B_i и B_j (i, j = 1, 2, ..., n) не существует экземпляра, типом которого одновременно являются типы сущности B_i и B_j.



У подтипа АЭРОПЛАН связь ПИЛОТ – унаследованная.

Тип сущности, на основе которого определяются подтипы, называется *супертипом*. Подтипы должны образовывать полное множество, т. е. любой экземпляр супертипа должен относиться к некоторому подтипу. Иногда для обеспечения такой полноты определяют дополнительный подтип ПРОЧИЕ.

ВЗАИМНО ИСКЛЮЧАЮЩИЕ СВЯЗИ

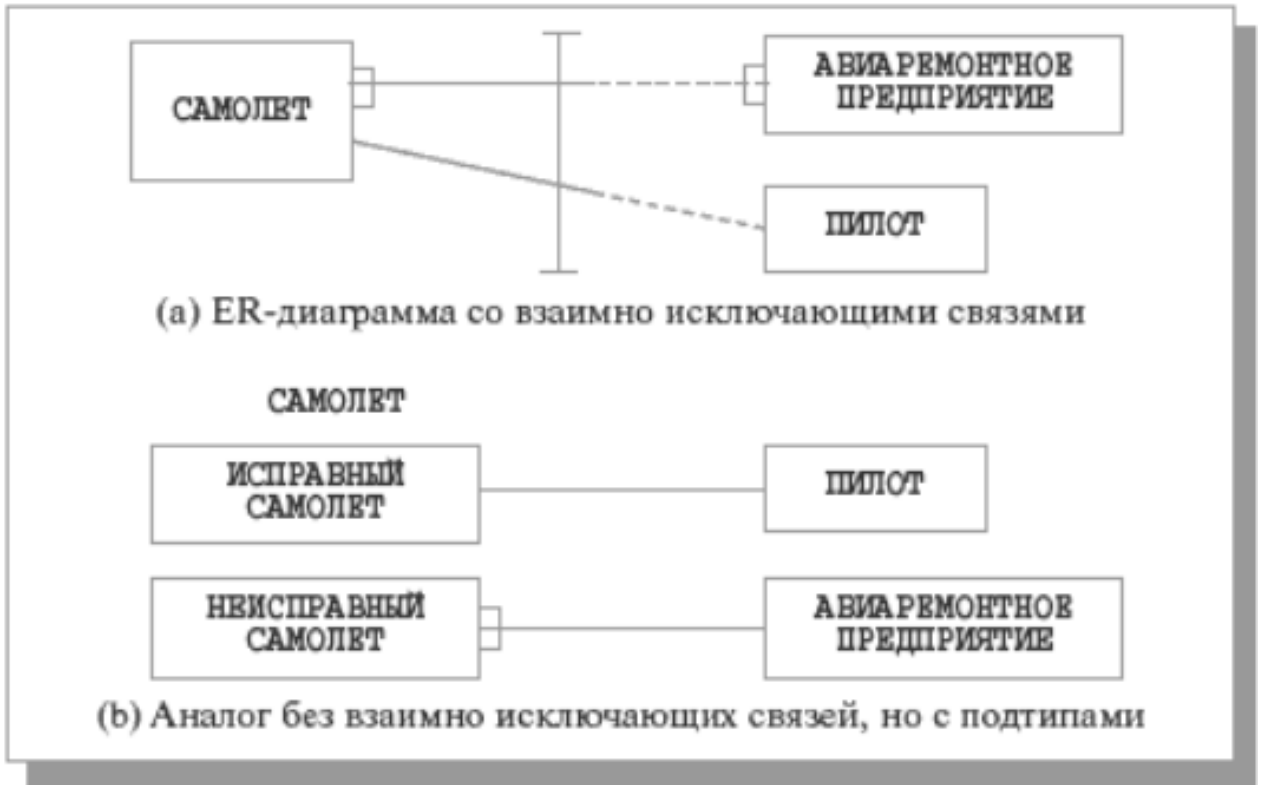
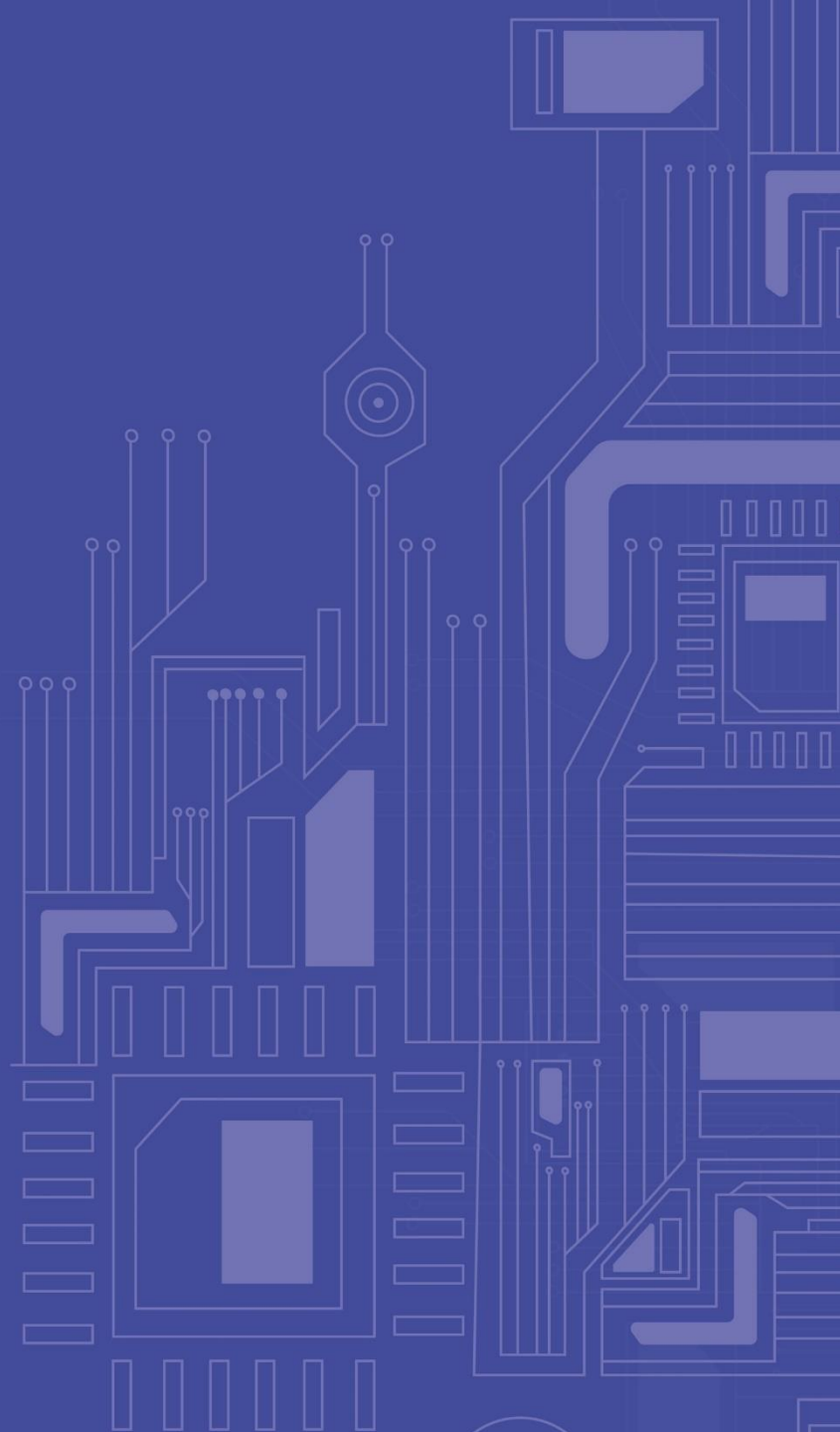


Диаграмма со взаимно исключающими связями может быть преобразована к диаграмме без взаимно исключающих связей путем введения подтипов. Поскольку любой самолет может быть либо исправным, либо неисправным, можно корректным образом ввести два подтипа супертипа САМОЛЕТ – ИСПРАВНЫЙ САМОЛЕТ и НЕИСПРАВНЫЙ САМОЛЕТ. На уровне супертипа сущности связи не определяются.

Для того чтобы описанная схема реализации механизма взаимно исключающих связей на основе механизма наследования действительно могла работать, в средствах манипулирования данными ER-модели должна быть предусмотрена возможность динамического изменения типа сущности у экземпляра.

ПОЛУЧЕНИЕ РЕЛЯЦИОННОЙ СХЕМЫ ИЗ ER-ДИАГРАММЫ



БАЗОВЫЕ ПРИНЦИПЫ (1/2)

1. Каждый простой тип сущности превращается в таблицу. (Простым типом сущности называется тип сущности, не являющийся подтипом и не имеющий подтипов.) Имя сущности становится именем таблицы. Экземплярам типа сущности соответствуют *строки* соответствующей таблицы.

2. Каждый атрибут становится столбцом таблицы с тем же именем; может выбираться более точный формат представления данных. Столбцы, соответствующие необязательным атрибутам, могут содержать неопределенные значения; столбцы, соответствующие обязательным атрибутам, – не могут.

3. Компоненты уникального идентификатора сущности превращаются в *первичный ключ таблицы*. Если в состав уникального идентификатора входят связи, к числу столбцов первичного ключа добавляется копия уникального идентификатора сущности, находящейся на дальнем конце связи.

4. Связи «многие к одному» (и «один к одному») становятся внешними ключами, т. е. образуется копия уникального идентификатора сущности на конце связи «один», и соответствующие столбцы составляют внешний ключ таблицы, соответствующей типу сущности на конце связи «многие».

БАЗОВЫЕ ПРИНЦИПЫ (2/2)

5 Чтобы отразить в определении таблицы ограничение, которое заключается в том, что степень конца связи должна равняться единице, соответствующий (возможно, составной) столбец должен быть дополнительно специфицирован как возможный ключ таблицы (в случае использования языка SQL для этого служит спецификация UNIQUE).

6 Для поддержки связи «многие ко многим» между типами сущности A и B создается дополнительная таблица AB с двумя столбцами, один из которых содержит уникальные идентификаторы экземпляров сущности A, а другой – уникальные идентификаторы экземпляров сущности B. Обозначим через УИД(c) уникальный идентификатор экземпляра с некоторого типа сущности C. Тогда, если в экземпляре связи «многие ко многим» участвуют экземпляры a_1, a_2, ..., a_n типа сущности A и экземпляры b_1, b_2, ..., b_m типа сущности B, то в таблице AB должны присутствовать все строки вида {УИД(a_i), УИД(b_j)} для $i = 1, 2, ..., n$, $j = 1, 2, ..., m$. Понятно, что, используя таблицы A, B и AB, с помощью стандартных реляционных операций можно найти все пары экземпляров типов сущности, участвующих в данной связи.

7 Индексы создаются для первичного ключа (уникальный индекс), внешних ключей и тех атрибутов, на которых предполагается в основном базировать запросы.

ПРЕДСТАВЛЕНИЕ В РЕЛЯЦИОННОЙ СХЕМЕ СУПЕРТИПОВ И ПОДТИПОВ СУЩНОСТЕЙ (1/2)

Есть два способа

Представление супертипа с подтипами в одной таблице

Таблица создается для максимального супертипа (типа сущности, не являющегося подтипом), а для подтипов могут создаваться представления. Таблица содержит столбцы, соответствующие каждому атрибуту (и связям) каждого подтипа. В таблицу добавляется, по крайней мере, один столбец, содержащий код ТИПА; он становится частью первичного ключа. Для каждой строки таблицы значение этого столбца определяет тип сущности, экземпляру которого соответствует строка. Столбцы этой строки, которые соответствуют атрибутам и связям, отсутствующим в данном типе сущности, должны содержать неопределенные значения.

Создание отдельной таблицы для каждого подтипа

При использовании этого метода для каждого подтипа первого уровня (для более глубоких уровней применяется первый метод) супертип воссоздается с помощью представления UNION (из всех таблиц подтипов выбираются общие столбцы – столбцы супертипа).

ПРЕДСТАВЛЕНИЕ В РЕЛЯЦИОННОЙ СХЕМЕ СУПЕРТИПОВ И ПОДТИПОВ СУЩНОСТЕЙ (2/2)

Достоинства и недостатки

Представление супертипа с подтипами в одной таблице

+

- Логично хранить вместе все строки, соответствующие экземплярам супертипа, поскольку любой экземпляр любого подтипа является экземпляром супертипа;
- Обеспечение простого доступа к экземплярам супертипа и к экземплярам подтипов;
- Возможность обойтись небольшим числом таблиц.

–

- Приложение, работающее с одной таблицей супертипа, должно иметь логику работы с разными наборами столбцов (по типам) и разными ограничениями целостности (в зависимости от связей);
- Общая для всех подтипов таблица – плохо при многопользовательском доступе;
- Потенциально в общей таблице будет много неопределенных значений, что может потребовать много внешней памяти.

Создание отдельной таблицы для каждого подтипа

+

- Более понятный метод – каждому подтипу соответствует таблица;
- Обеспечение простого доступа к экземплярам супертипа и к экземплярам подтипов;
- Упрощается логика приложений. Каждая подпрограмма работает со своей таблицей.

–

- В общем случае требуется слишком много отдельных таблиц;
- Работа с экземплярами супертипа на основе представления, объединяющего таблицы супертипов, может оказаться недостаточно эффективной;
- Поскольку множество экземпляров супертипа является объединением множеств экземпляров подтипов, не все РСУБД могут обеспечить выполнение операций модификации экземпляров супертипа.

ПРЕДСТАВЛЕНИЕ В РЕЛЯЦИОННОЙ СХЕМЕ ВЗАИМНО ИСКЛЮЧАЮЩИХ СВЯЗЕЙ

Три способа представления:

- Общее хранение внешних ключей
- Раздельное хранение внешних ключей
- Выявление общего типа сущностей

Особенности разных способов представления

id судна	название	состояние	общий внешний ключ	внешний ключ пилота	внешний ключ ангара
1	самолет	пилот	id_пилота1	id_пилота1	NULL
2	вертолет	ремонт	id_ангара1	NULL	id_ангара1
3	самолет	пилот	id_пилота1	id_пилота1	
4	самолет	ремонт	id_ангара1	NULL	id_ангара1
5	самолет	ремонт	id_ангара2	NULL	id_ангара2
6	самолет	пилот	id_пилота1	id_пилота1	NULL
7	самолет	пилот	id_пилота2	id_пилота2	NULL
8	вертолет	пилот	id_пилота1	id_пилота1	NULL
9	самолет	ремонт	id_ангара2	NULL	id_ангара2

Странно искать общее у пилота и ангара, но если вдруг, то проблема отменяется.

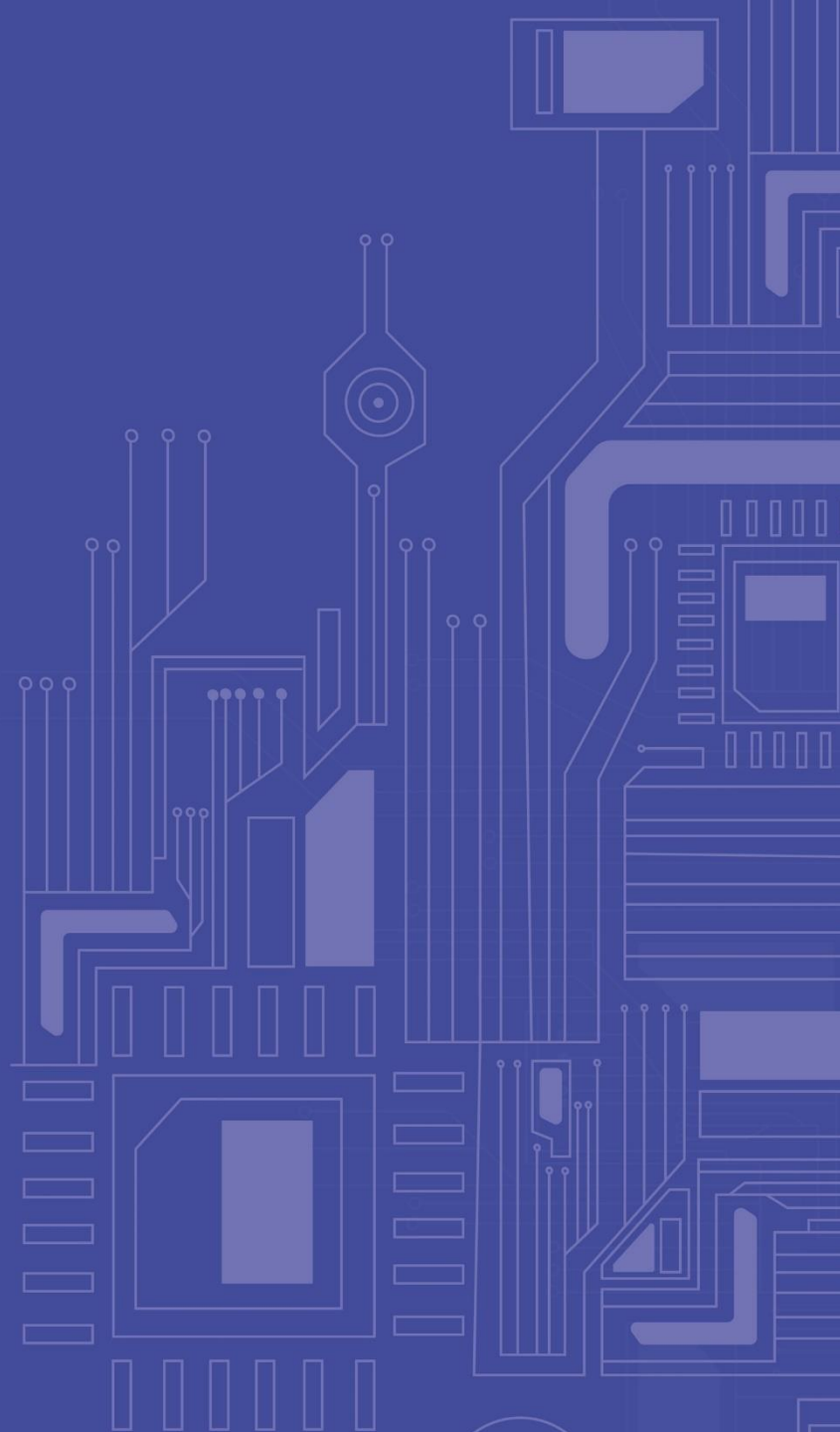
Возможные варианты избежать взаимно исключающих сущностей

(a) ER-диаграмма с взаимно исключающими связями

(b) ER-диаграмма с подтипами сущности без альтернативных связей

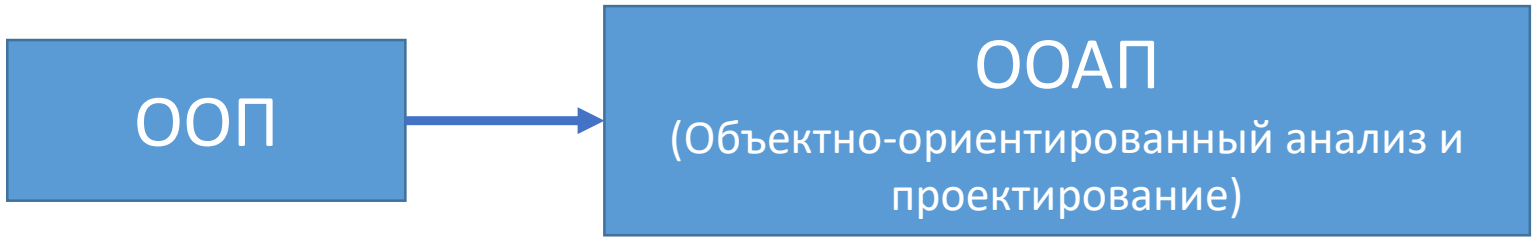
(c) ER-диаграмма, в которой альтернативные связи заменены связью с супертипом сущности

ДИАГРАММЫ КЛАССОВ ЯЗЫКА UML



ЯЗЫК UML (Unified Modeling Language)

UML позволяет моделировать разные виды систем: чисто программные, чисто аппаратные, программно-аппаратные, смешанные, явно включающие деятельность людей и т. д.

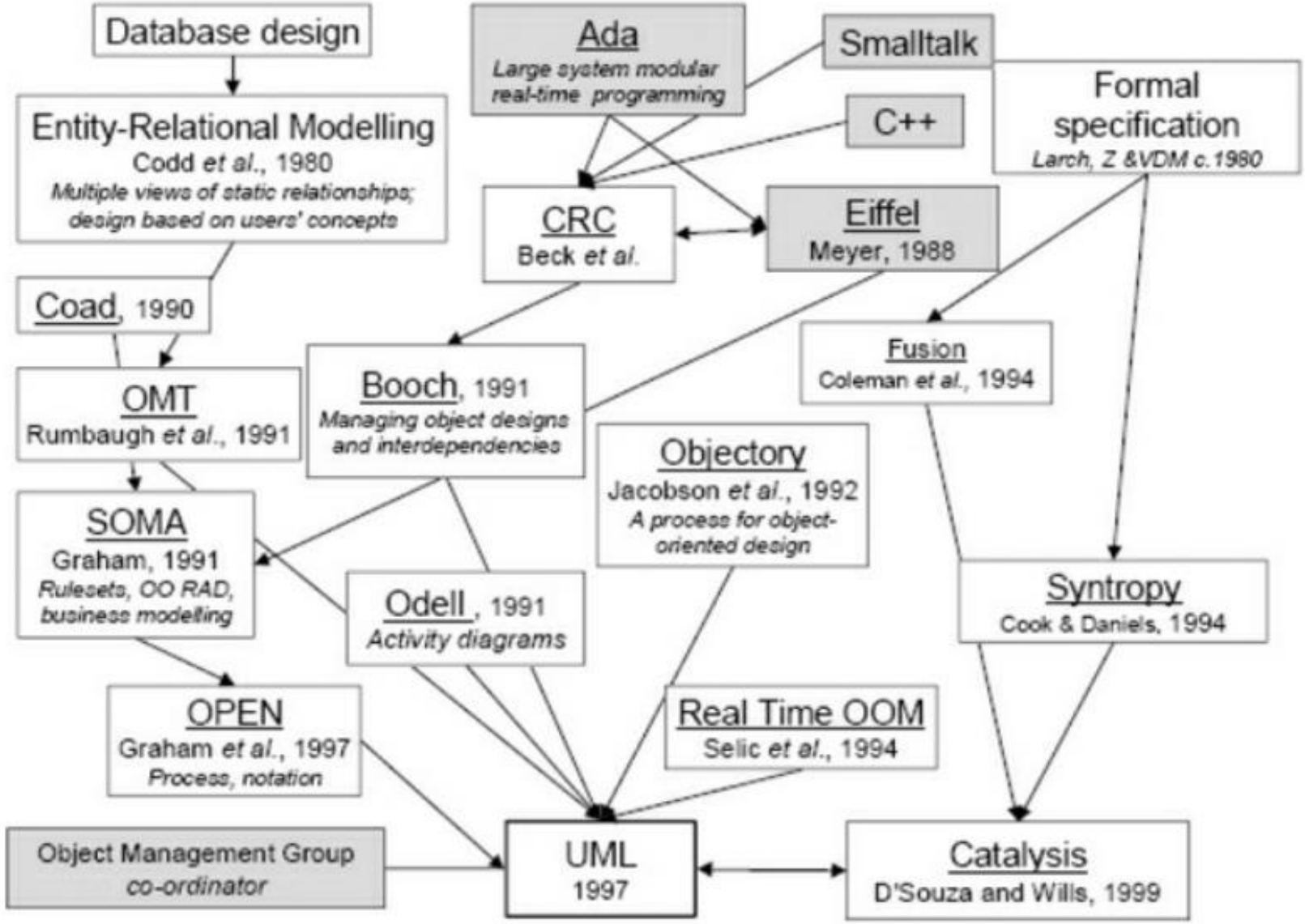


Прежде, чем начать *программирование классов*, их свойств и методов, необходимо определить сами эти *классы*. Более того, нужно дать ответы на следующие вопросы: сколько и какие *классы* нужно определить для решения поставленной задачи, какие свойства и методы необходимы для придания *классам* требуемого поведения, а также установить взаимосвязи между *классами*. Эта совокупность задач не столько связана с написанием кода, сколько с общим анализом требований к будущей программе, а также с анализом конкретной *предметной области*, для которой разрабатывается *программа*.

Язык UML активно применяется для проектирования реляционных БД. Для этого используется небольшая часть языка (диаграммы классов), да и то не в полном объеме. С точки зрения проектирования реляционных БД модельные возможности не слишком отличаются от возможностей ER-диаграмм.

ИСТОРИЯ ЯЗЫКА UML

Отдельные языки *объектно-ориентированного моделирования* начали появляться в середине 1970-х годов, когда различные исследователи и программисты предлагали свои подходы к ООАП. В период между 1989 -1994 гг. общее число наиболее известных языков моделирования возросло с 10 до более чем 50.
В январе 1997 года был опубликован документ с описанием языка *UML* 1.0



НОТАЦИИ ЯЗЫКА UML И ВИДЫ ДИАГРАММ

Диаграмма (diagram) — графическое представление совокупности элементов *модели* в форме связного графа, вершинам и ребрам (дугам) которого приписывается определенная семантика.

Нотация ***канонических диаграмм*** - основное средство разработки *моделей* на языке UML.

В нотации языка *UML* определены следующие виды *канонических диаграмм*:
вариантов использования (use case diagram)

- классов (*class diagram*)
- кооперации (collaboration diagram)
- последовательности (*sequence diagram*)
- состояний (statechart diagram)
- деятельности (*activity diagram*)
- компонентов (component diagram)
- развертывания (deployment diagram)

СЛОЖНАЯ СИСТЕМА В UML

Каждая из этих *диаграмм* детализирует и конкретизирует различные представления о *модели* сложной системы в терминах языка *UML*. При этом *диаграмма* вариантов использования представляет собой наиболее общую концептуальную *модель* сложной системы, которая является исходной для построения всех остальных *диаграмм*.

Диаграмма классов, *по* своей сути, логическая *модель*, отражающая статические аспекты структурного построения сложной системы.
Диаграммы кооперации (взаимодействия) и последовательностей представляют собой разновидности логической *модели*, которые отражают динамические аспекты функционирования сложной системы.

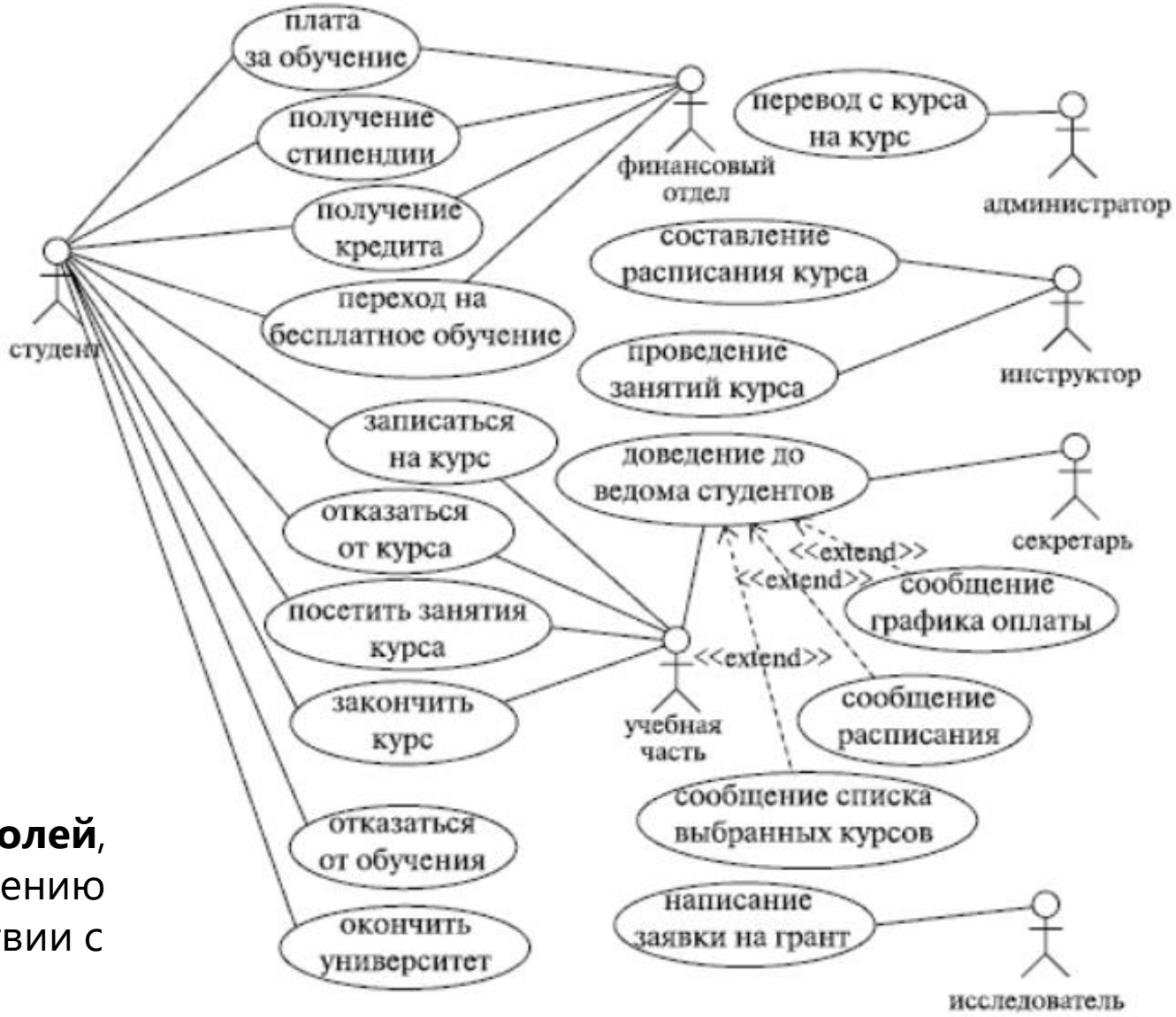
Диаграммы состояний и деятельности предназначены для моделирования поведения системы. И, наконец, *диаграммы* компонентов и развертывания служат для представления физических компонентов сложной системы и поэтому относятся к ее физической *модели*.

ДИГРАММА ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ

Диаграмма прецедентов/ (use case diagram)

Вариант использования представляет собой спецификацию общих особенностей поведения или функционирования моделируемой системы без рассмотрения внутренней структуры этой системы. Несмотря на то, что каждый *вариант использования* определяет последовательность действий, которые должны быть выполнены проектируемой системой при взаимодействии ее с соответствующим *актером*, сами эти действия не изображаются на рассматриваемой диаграмме.

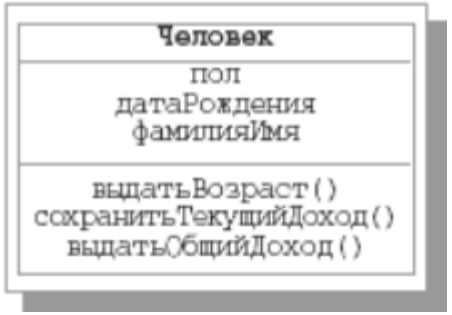
Актер (actor) — согласованное множество **ролей**, которые играют *внешние сущности* по отношению к *вариантам использования* при взаимодействии с ними.

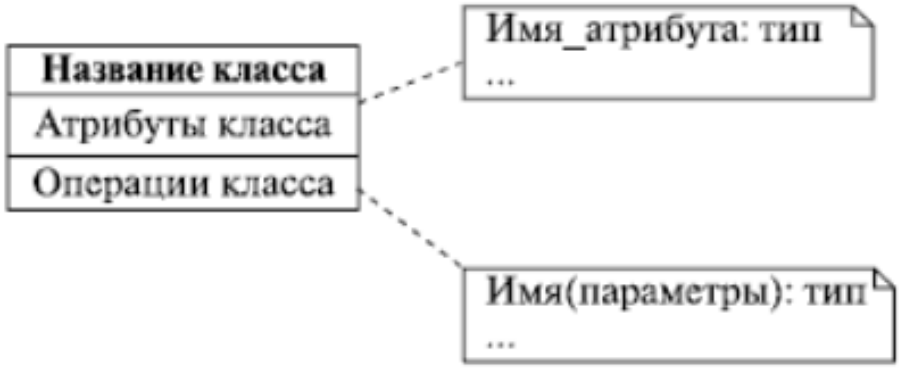


КЛАССЫ, АТРИБУТЫ, ОПЕРАЦИИ

Классом называется именованное описание совокупности объектов с общими атрибутами, операциями, связями и семантикой.

Атрибутом класса называется именованное свойство класса, описывающее множество значений, которые могут принимать экземпляры этого свойства.





Операцией класса называется именованная услуга, которую можно запросить у любого объекта этого класса.

Символ	Значение
+	public - открытый доступ
-	private - только из операций того же класса
#	protected - только из операций этого же класса и классов, создаваемых на его основе

КАТЕГОРИИ СВЯЗЕЙ

В диаграмме классов могут участвовать связи трех разных категорий:

- зависимость (dependency),
- обобщение (generalization),
- ассоциация (association).

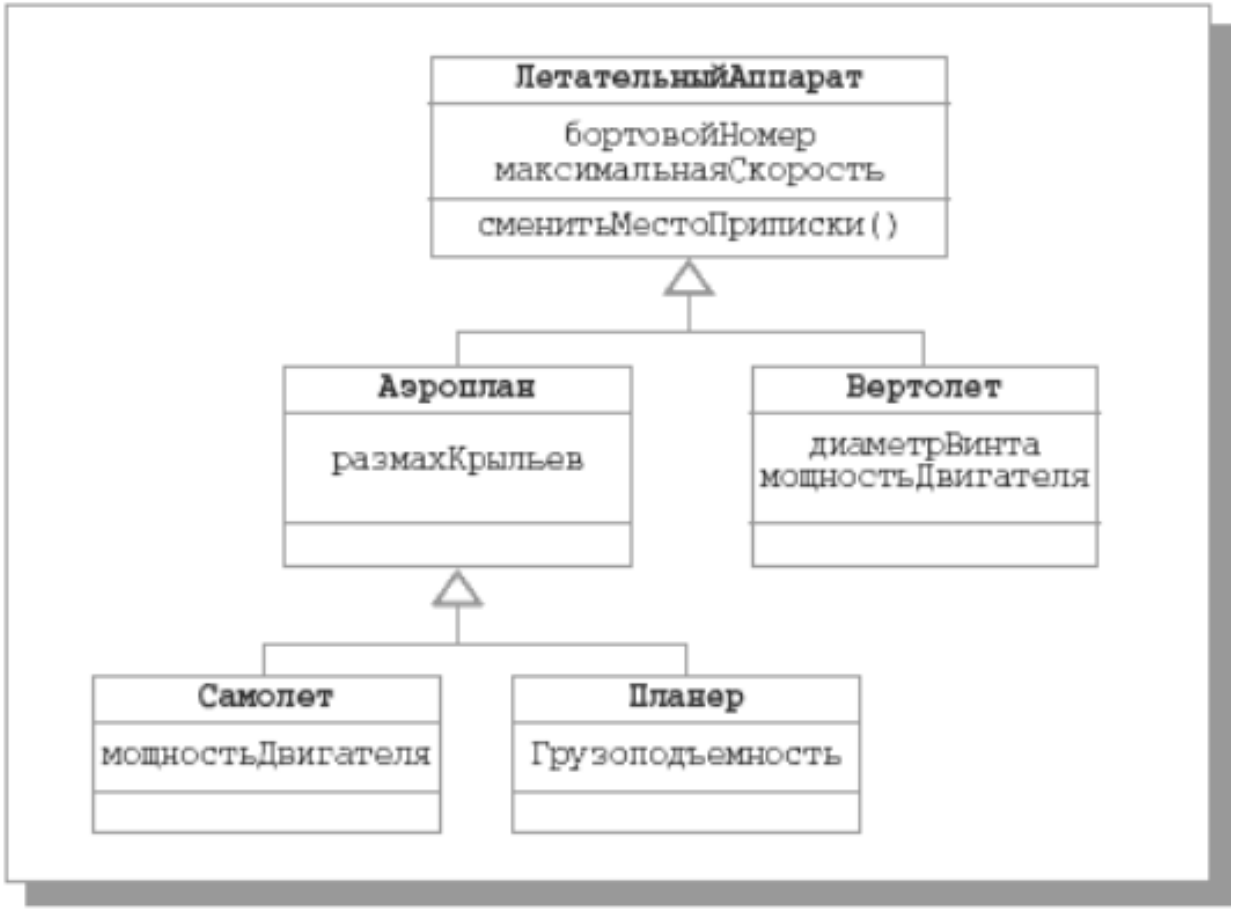
Зависимостью называют связь по применению, когда изменение в спецификации одного класса может повлиять на поведение другого класса, использующего первый класс. Чаще всего зависимости применяют в диаграммах классов, чтобы отразить в сигнатуре операции одного класса тот факт, что параметром этой операции могут быть объекты другого класса.

Зависимость показывается прерывистой линией со стрелкой, направленной к классу, от которого имеется зависимость. Очевидно, что связи-зависимости существенны для объектно-ориентированных систем (в том числе и для ООБД).

При проектировании реляционных БД непонятно, что делать с зависимостями (как воспользоваться этой информацией в реляционной БД?).

СВЯЗИ-ОБОБЩЕНИЯ И МЕХАНИЗМ НАСЛЕДОВАНИЯ В UML (1/3)

Связью-обобщением называется связь между общей сущностью, называемой *суперклассом*, или родителем, и более специализированной разновидностью этой сущности, называемой *подклассом*, или потомком.



Объекты класса-потомка могут использоваться везде, где могут использоваться объекты класса-предка. Это свойство называют ***полиморфизмом по включению***, имея в виду, что объекты потомка можно считать включаемыми во множество объектов класса-предка.

Графически обобщения изображаются в виде сплошной линии с большой незакрашенной стрелкой, направленной к суперклассу.

На рисунке показан пример иерархии ***одиночного наследования***: у каждого подкласса имеется только один суперкласс.

СВЯЗИ-ОБОБЩЕНИЯ И МЕХАНИЗМ НАСЛЕДОВАНИЯ В UML (2/3)

В UML допускается ***множественное наследование***, когда один подкласс определяется на основе нескольких суперклассов.

Пример:

Среди объектов класса Студент могут быть преподаватели, а некоторые преподаватели могут быть студентами. Тогда мы можем определить класс *СтудентПреподаватель* путем множественного наследования от суперклассов Студент и Преподаватель. Объект класса *СтудентПреподаватель* обладает всеми свойствами и операциями классов Студент и Преподаватель и может быть использован везде, где могут применяться объекты этих классов. Так что полиморфизм по включению продолжает работать.

Существует проблема именования атрибутов и операций в подклассе, полученном путем множественного наследования. Например, предположим, что при образовании подклассов Студент и Преподаватель в них обоих был определен атрибут с именем *НомерКомнаты*. Очень вероятно, что для объектов класса Студент значениями этого атрибута будут номера комнат в студенческом общежитии, а для объектов класса Преподаватель – номера служебных кабинетов.

СВЯЗИ-ОБОБЩЕНИЯ И МЕХАНИЗМ НАСЛЕДОВАНИЯ В UML (3/3)

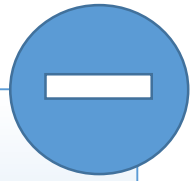
Пример:

Предположим, что при образовании подклассов Студент и Преподаватель в них обоих был определен атрибут с именем *НомерКомнаты*.

Как быть с объектами класса *СтудентПреподаватель*, для которых существенны оба одноименных атрибута (у студента-преподавателя могут иметься и комната в общежитии, и служебный кабинет)?

На практике применяется одно из следующих решений:

1. Запретить образование подкласса СтудентПреподаватель, пока в одном из суперклассов не будет произведено переименование атрибута *НомерКомнаты*;
2. Наследовать это свойство только от одного из суперклассов, так что, например, значением атрибута *НомерКомнаты* у объектов класса СтудентПреподаватель всегда будут номера служебных кабинетов;
3. Унаследовать в подклассе оба свойства, но автоматически переименовать оба атрибута, чтобы прояснить их смысл; назвать их, например, *НомерКомнатыСтудента* и *НомерКомнатыПреподавателя*.



1. Требует возврата к ранее определенному классу, имена атрибутов и операций которого, возможно, уже используются в приложениях.
2. Нарушает логику наследования, не давая возможности на уровне подкласса использовать все свойства суперклассов.
3. Заставляет использовать длинные имена атрибутов и операций, которые могут стать недопустимо длинными, если процесс множественного наследования будет продолжаться .

СВЯЗИ-АССОЦИАЦИИ

Ассоциацией называется структурная связь, показывающая, что объекты одного класса некоторым образом связаны с объектами другого или того же самого класса. Допускается, чтобы оба конца ассоциации относились к одному классу. В ассоциации могут связываться два класса, и тогда она называется *бинарной*. Допускается создание ассоциаций, связывающих сразу ***n*** классов (они называются *n-арными ассоциациями*).

Ассоциации может быть присвоено *имя*, характеризующее природу связи. Смысл имени уточняется с помощью черного треугольника, который располагается над линией связи справа или слева от имени ассоциации. Этот треугольник указывает направление чтения имя связи.

Роль класса, как и имя конца связи в ER-модели, задается именем, помещаемым под линией ассоциации ближе к данному классу.

В общем случае, для ассоциации могут задаваться и ее собственное имя, и имена ролей классов. Это связано с тем, что класс может играть одну и ту же роль в разных ассоциациях, так что в общем случае пара имен ролей классов не идентифицирует ассоциацию.

КРАТНОСТЬ АССОЦИАЦИИ

Кратностью (multiplicity) роли ассоциации называется характеристика, указывающая, сколько объектов класса с данной ролью может или должно участвовать в каждом экземпляре ассоциации

«1» - каждый объект класса с данной ролью должен участвовать в некотором экземпляре данной ассоциации, причем в каждом экземпляре ассоциации может участвовать ровно один объект класса с данной ролью.

«0..1» - не все объекты класса с данной ролью обязаны участвовать в каком-либо экземпляре данной ассоциации, но в каждом экземпляре ассоциации может участвовать только один объект.

«1..*» - все объекты класса с данной ролью должны участвовать в некотором экземпляре данной ассоциации, и в каждом экземпляре ассоциации должен участвовать хотя бы один объект (верхняя граница не задана).

В более сложных (но крайне редко встречающихся на практике) случаях определения кратности можно использовать списки диапазонов. Например, список «2, 4..6, 8..*» говорит о том, что все объекты класса с указанной ролью должны участвовать в некотором экземпляре данной ассоциации, и в каждом экземпляре ассоциации должны участвовать два, от четырех до шести или более семи объектов класса с данной ролью.

АГРЕГАТНЫЕ АССОЦИАЦИИ

Иногда в диаграмме классов требуется отразить тот факт, что ассоциация между двумя классами имеет специальный вид «часть-целое». В этом случае класс «целое» имеет более высокий концептуальный уровень, чем класс «часть». Ассоциация такого рода называется *агрегатной*. Графически агрегатные ассоциации изображаются в виде простой ассоциации с незакрашенным ромбом на стороне класса-«целого».

Бывают случаи, когда связь «части» и «целого» настолько сильна, что уничтожение «целого» приводит к уничтожению всех его «частей». Агрегатные ассоциации, обладающие таким свойством, называются *композитными*, или просто *композициями*. При наличии композиции объект-часть может быть частью только одного объекта-целого (композита). При обычной агрегатной ассоциации «часть» может одновременно принадлежать нескольким «целым». Графически композиция изображается в виде простой ассоциации, дополненной закрашенным ромбом со стороны «целого».

В контексте проектирования реляционных БД агрегатные и в особенности композитные ассоциации влияют только на способ поддержки ссылочной целостности. В частности, композитная связь является явным указанием на то, что ссылочная целостность между «целым» и «частями» должна поддерживаться путем каскадного удаления частей при удалении целого.

НАВИГАЦИЯ ПО АССОЦИАЦИИ

При наличии простой ассоциации между двумя классами предполагается возможность навигации между объектами, входящими в один экземпляр ассоциации. Если не оговорено иное, то навигация по ассоциации может проводиться в обоих направлениях. Однако бывают случаи, когда желательно ограничить направление навигации для некоторых ассоциаций. В этом случае на линии ассоциации ставится стрелка, указывающая направление навигации.

С точки зрения реляционных БД ассоциации с однонаправленной навигацией можно считать указанием на необходимость ограничения видимости объектов БД. Соответствующую (но существенно более общую) возможность в SQL-ориентированных БД обеспечивает механизм представлений (view).

РЕКОМЕНДАЦИИ ИСПОЛЬЗОВАНИЯ ЯЗЫКА UML (1/2)

- Каждая *диаграмма* должна служить законченным представлением соответствующего фрагмента моделируемой предметной области. Речь идет о том, что в процессе разработки *диаграммы* необходимо учесть все сущности, важные с точки зрения контекста данной *модели* и *диаграммы*. Отсутствие тех или иных элементов на *диаграмме* служит признаком неполноты *модели* и может потребовать ее последующей доработки.
- Все сущности на *диаграмме модели* должны быть одного *уровня представления*. Здесь имеется в виду согласованность не только имен одинаковых элементов, но и возможность вложения отдельных *диаграмм* друг в друга для достижения полноты представлений. В случае достаточно сложных *моделей* систем желательно придерживаться стратегии последовательного уточнения или детализации отдельных *диаграмм*.
- Вся информация о сущностях должна быть явно представлена на *диаграммах*. В языке UML при отсутствии некоторых символов на *диаграмме* могут быть использованы их значения по умолчанию (например, в случае неявного указания видимости атрибутов и операций классов), тем не менее, необходимо стремиться к явному указанию свойств всех элементов *диаграмм*.
- *Диаграммы* не должны содержать противоречивой информации. Противоречивость *модели* может служить причиной серьезных проблем при ее реализации и последующем использовании на практике. Например, наличие замкнутых путей при изображении отношений агрегирования или композиции приводит к ошибкам в программном коде, который будет реализовывать соответствующие классы. Наличие элементов с одинаковыми именами и различными атрибутами свойств в одном пространстве имен также приводит к неоднозначной интерпретации и может быть источником проблем.

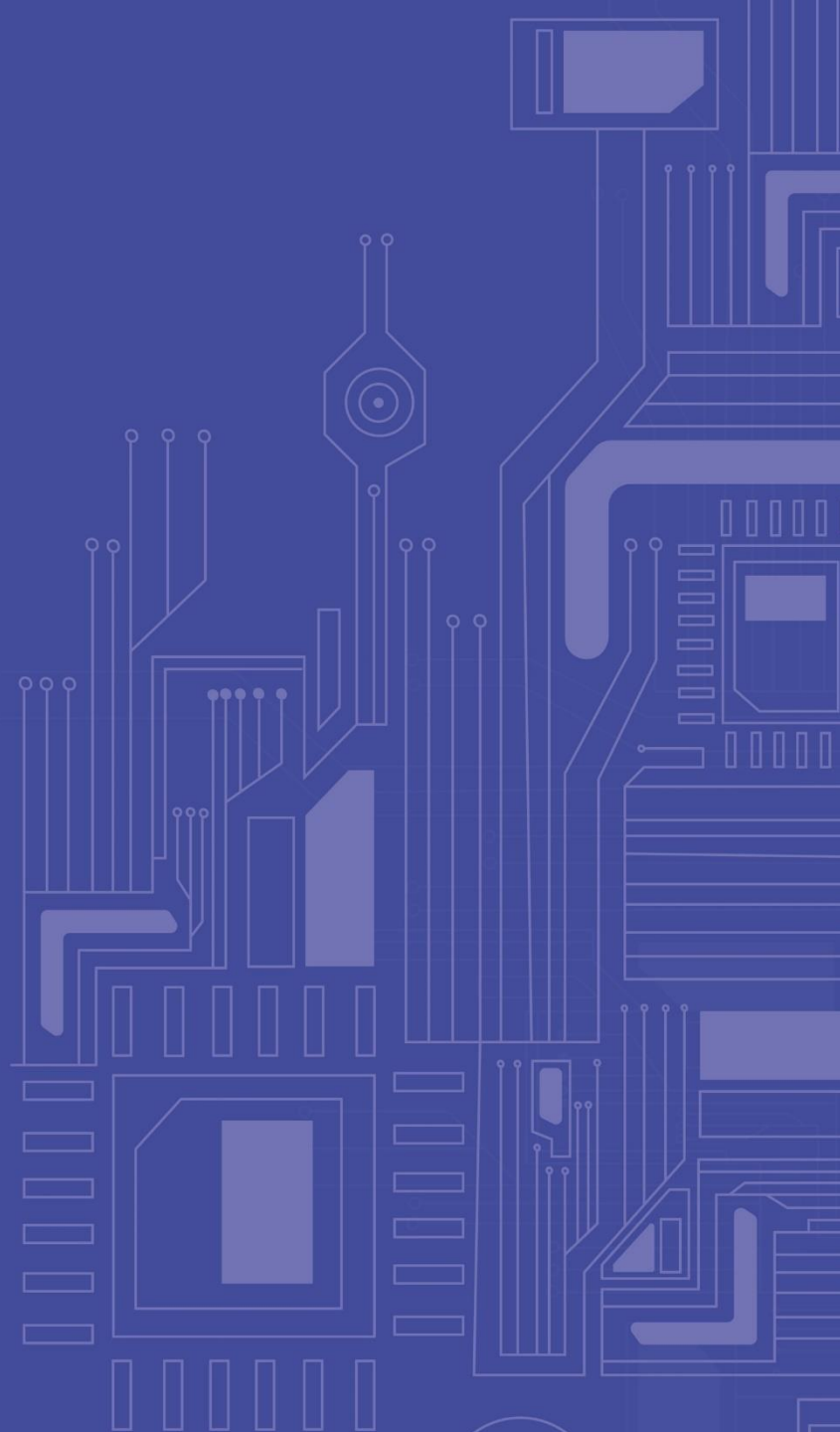
(Рисуем всё, обобщаем и раскрываем последовательно, отображаем всю информацию, не противоречим себе.)

РЕКОМЕНДАЦИИ ИСПОЛЬЗОВАНИЯ ЯЗЫКА UML (2/2)

- Каждая *диаграмма* должна быть самодостаточной для правильной интерпретации всех ее элементов и понимания семантики всех используемых графических символов. Любые пояснительные тексты, которые не являются собственными элементами *диаграммы* (например, комментариями), не должны приниматься во внимание разработчиками. В то же время общие фрагменты *диаграмм* могут уточняться или детализироваться на других *диаграммах* этого же типа, образуя вложенные или подчиненные *диаграммы*. Таким образом, *модель* системы на языке UML представляет собой *пакет* иерархически вложенных *диаграмм*, детализация которых должна быть достаточной для последующей генерации программного кода, реализующего проект соответствующей системы.
- Количество типов *диаграмм* для конкретной *модели* приложения строго не фиксировано. Для простых приложений нет необходимости строить все без исключения типы *диаграмм*. Некоторые из них могут просто отсутствовать в проекте системы, и это не будет считаться ошибкой разработчика. Например, *модель системы* может не содержать *диаграмму* развертывания для приложения, выполняемого локально на компьютере пользователя. Важно понимать, что перечень *диаграмм* зависит от специфики конкретного проекта системы.
- Любая *модель системы* должна содержать только те элементы, которые определены в нотации языка UML. Имеется в виду требование начинать разработку проекта, используя только те конструкции, которые уже определены в *метамодели* UML. Как показывает практика, этих конструкций вполне достаточно для представления большинства типовых проектов программных систем. И только при отсутствии необходимых базовых элементов языка UML следует использовать механизмы их расширения для адекватного представления конкретной *модели системы*. Не допускается переопределение семантики тех элементов, которые отнесены к базовой нотации *метамодели* языка UML.

(Меньше слов, больше картинок и диаграмм, не надо придумывать свое.)

ОГРАНИЧЕНИЯ ЦЕЛОСТНОСТИ И ЯЗЫК OCL



ВЫРАЖЕНИЕ ОГРАНИЧЕНИЙ

В диаграммах классов могут указываться ограничения целостности, которые должны поддерживаться в проектируемой БД. В UML допускаются два способа определения ограничений: на естественном языке и на языке OCL. Более точный и лаконичный способ формулировки ограничений обеспечивает язык OCL (Object Constraints Language).

Выражение ограничения на естественном языке

(Студент стипендия должно находиться в диапазоне между Университет.минСтипендия и Университет.максСтипендия)

Под ***инвариантом класса*** в OCL понимается условие, которому должны удовлетворять все объекты данного класса. Если говорить более точно, инвариант класса – это логическое выражение, вычисление которого должно давать *true* при создании любого объекта данного класса и сохранять истинное значение в течение всего времени существования этого объекта.

```
context <class_name> inv:
<OCL-выражение>
```

Здесь <class-name> является именем класса, для которого определяется инвариант, inv – ключевое слово, говорящее о том, что определяется именно инвариант, а не ограничение другого вида, и context – ключевое слово, которое говорит о том, что контекстом следующего после двоеточия OCL-выражения являются объекты класса <class-name>, т. е. OCL-выражение должно принимать значение true для всех объектов этого класса.

ОПЕРАЦИИ OCL (1/3)

- операции над значениями предопределенных в UML (скалярных) типов данных;
- операции над объектами;
- операции над множествами;
- операции над мультимножествами;
- операции над последовательностями.

Операции над объектами

В OCL определены три операции над объектами:

- получение значения атрибута;
- переход по соединению,
- вызов операции класса (последняя операция для целей проектирования реляционных БД несущественна).

```
<объект>.<имя атрибута>
```

```
<объект>.<имя роли, противоположенной по отношению к объекту>
```

ОПЕРАЦИИ OCL (2/3)

В OCL поддерживается обширный набор операций над значениями коллекционных типов данных. Синтаксически операции над коллекциями записываются в нотации, аналогичной точечной, но вместо точки используется стрелка (→).

```
<коллекция> ⟶ <имя операции> (<список фактических параметров>)
```

Операции над множествами, мультимножествами и последовательностями

- Операция select
- Операция collect
- Операции exists, forAll, size
- Операции union, intersect, symmetricDifference

UNION

Результатом операции **union**, определенной над множеством и мультимножеством, является мультимножество, т. е. из результата объединения таких двух коллекций дубликаты не исключаются. Результатом же операции **union**, определенной над двумя множествами, является множество, т. е. в этом случае возможные дубликаты должны быть исключены.

SELECT

В OCL определены три одноименных операции **select**, которые обрабатывают заданное множество, мультимножество или последовательность на основе заданного логического выражения над элементами коллекции. Результатом каждой операции является новое множество, мультимножество или последовательность, соответственно, из тех элементов входной коллекции, для которых результатом вычисления логического выражения является *true*.

ОПЕРАЦИИ OCL (3/3)

COLLECT

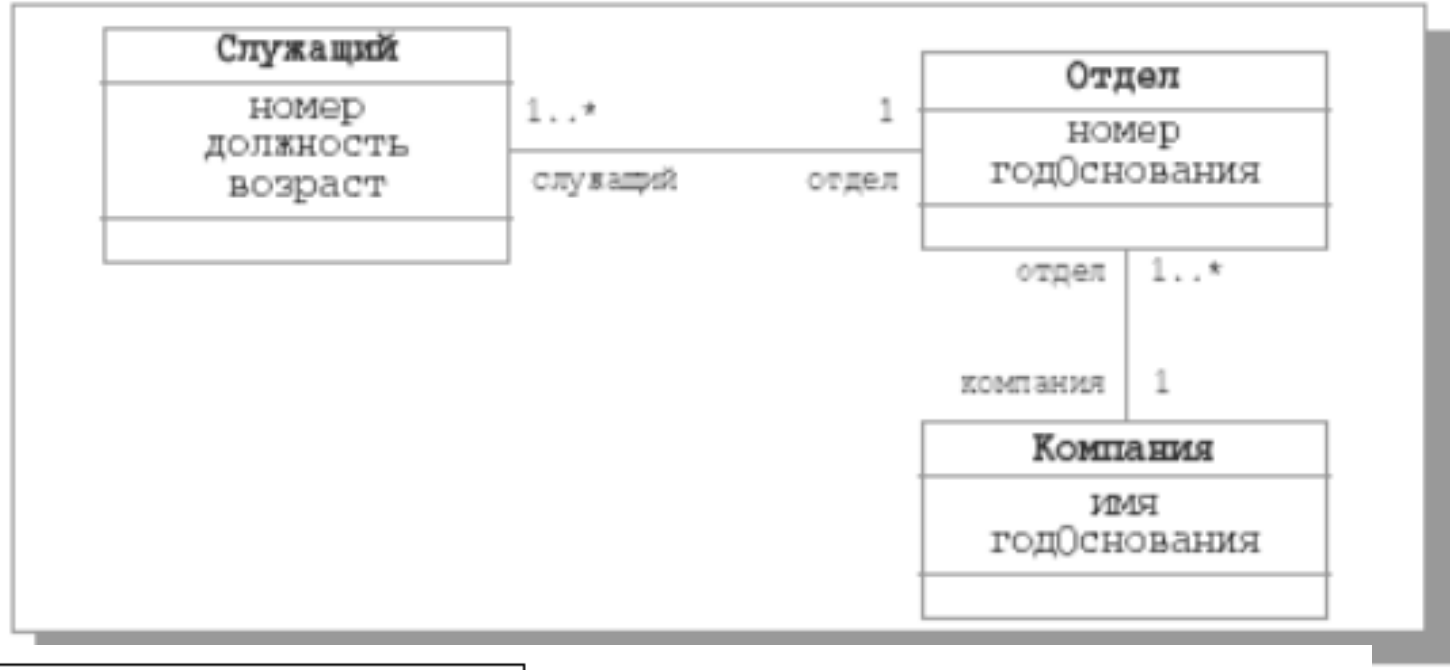
Аналогично набору операций **select**, в OCL определены три операции **collect**, параметрами которых являются множество, мультимножество или последовательность и некоторое выражение над элементами соответствующей коллекции. Результатом является мультимножество для операций **collect**, определенных над множествами и мультимножествами, и последовательность для операции **collect**, определенной над последовательностью. При этом результирующая коллекция соответствующего типа (коллекция значений или объектов) состоит из результатов применения выражения к каждому элементу входной коллекции. Операция **collect** используется, главным образом, в тех случаях, когда от заданной коллекции объектов требуется перейти к некоторой другой коллекции объектов, которые ассоциированы с объектами исходной коллекции через некоторое соединение. В этом случае выражение над элементом исходной коллекции основывается на операции перехода по соединению.

EXISTS, FORALL, SIZE

В OCL определены три одноименных операции **exists** над множеством, мультимножеством и последовательностью, дополнительным параметром которых является логическое выражение. В результате каждой из этих операций выдается *true* в том и только в том случае, когда хотя бы для одного элемента входной коллекции значением логического выражения является *true*. В противном случае результатом операции является *false*. Операции **forAll** отличаются от операций **exists** тем, что в результате каждой из них выдается true в том и только в том случае, когда для всех элементов входной коллекции результатом вычисления логического выражения является *true*. В противном случае результатом операции будет *false*. Операция **size** применяется к коллекции и выдает число содержащихся в ней элементов.

ПРИМЕРЫ ИНВАРИАНТОВ (1/4)

1. Определить ограничение «возраст служащих должен быть больше 18 и меньше 100 лет».

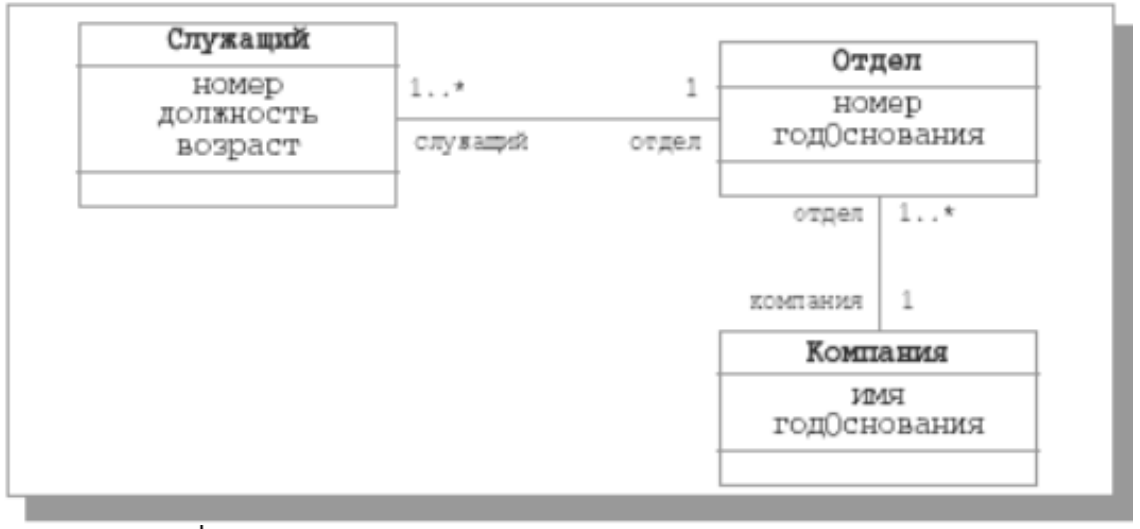



```
context Служащий inv:
self.возраст > 18 and self.возраст < 100
```

Условие инварианта накладывает требуемое ограничение на значения атрибута `возраст`, определенного в классе `Служащий`. В условном выражении инварианта ключевое слово `self` обозначает текущий объект класса-контекста инварианта. Можно считать, что при проверке данного условия будут перебираться существующие объекты класса `Служащий`, и для каждого объекта будет проверяться, что значения атрибута `возраст` находятся в пределах заданного диапазона. Ограничение удовлетворяется, если условное выражение принимает значение `true` для каждого объекта класса-контекста.

ПРИМЕРЫ ИНВАРИАНТОВ (2/4)

2. Выразить на языке OCL ограничение, в соответствии с которым в отделах с номерами больше 5 должны работать служащие старше 30 лет.

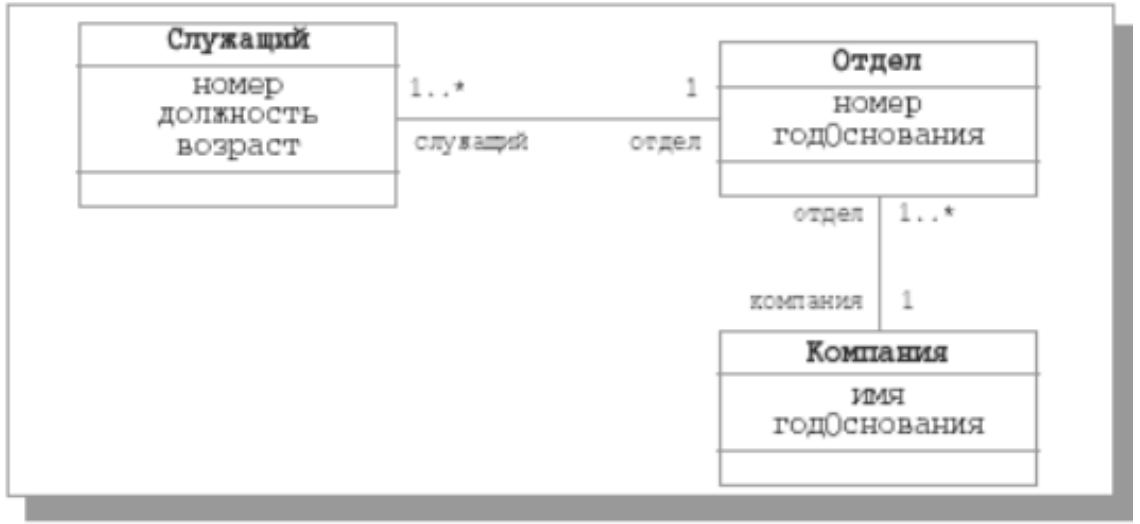



```
context Отдел inv:
self.номер ≤ 5 or
self.служащий ⟶ select (возраст ≤ 30) ⟶ size () = 0
```

В этом случае условное выражение инварианта будет вычисляться для каждого объекта класса `Отдел`. Подвыражение справа от операции `or` вычисляется слева направо. Сначала вычисляется подвыражение `self.служащий`, значением которого является множество объектов, соответствующих служащим, которые работают в текущем отделе. Далее к этому множеству применяется операция `select` (возраст 30), в результате которой вырабатывается множество объектов, соответствующих служащим текущего отдела, возраст которых не превышает 30 лет. Значением операции `size ()` является число объектов в этом множестве. Все выражение принимает значение `true`, если последняя операция сравнения «`=0`» вырабатывает значение `true`, т. е. если в текущем отделе нет служащих младше 31 года. Ограничение в целом удовлетворяется только в том случае, если значением условия инварианта является `true` для каждого отдела.

ПРИМЕРЫ ИНВАРИАНТОВ (3/4)

3. Определить ограничение, в соответствии с которым у каждого отдела должен быть менеджер, и любой отдел должен быть основан не раньше соответствующей компании.



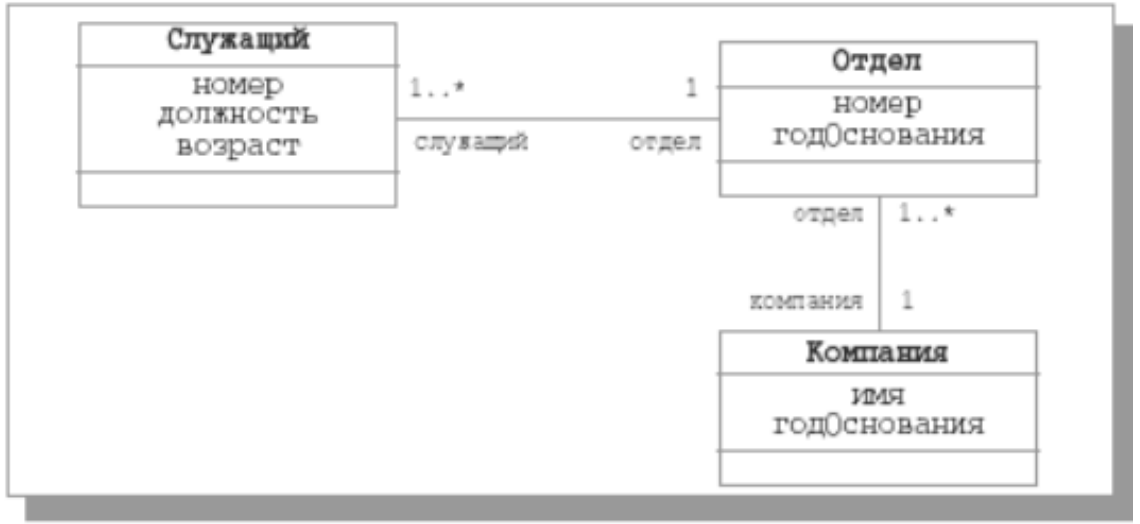

```
context Отдел inv:
self.служащий ⟶ exists (должность = "manager") and
self.компания.годОснования ≤ self.годОснования
```

Здесь должность – атрибут класса `Служащий`, а атрибуты с именем `годОснования` имеются и у класса `Отдел`, и у класса `Компания`. В условном выражении этого инварианта подвыражение `self.служащий ⟶ exists (должность = "manager")` эквивалентно выражению `self.служащий ⟶ select (должность = "manager") ⟶ size () > 1`. Если бы в ограничении мы потребовали, чтобы у каждого отдела был только один менеджер, то следовало бы написать `... size () = 1`, и это было бы не эквивалентно варианту с `exists`.

Обратите внимание, что в этом случае снова законным является подвыражение `self.компания.годОснования`, поскольку кратность роли `компания` в ассоциации классов `Отдел` и `Компания` равна единице.

ПРИМЕРЫ ИНВАРИАНТОВ (4/4)

4. Ограничить максимально возможное количество служащих компании числом 1000.




```
context Компания inv:
self.отдел ⟶ collect (служащие) ⟶ size ( ) < 1000
```

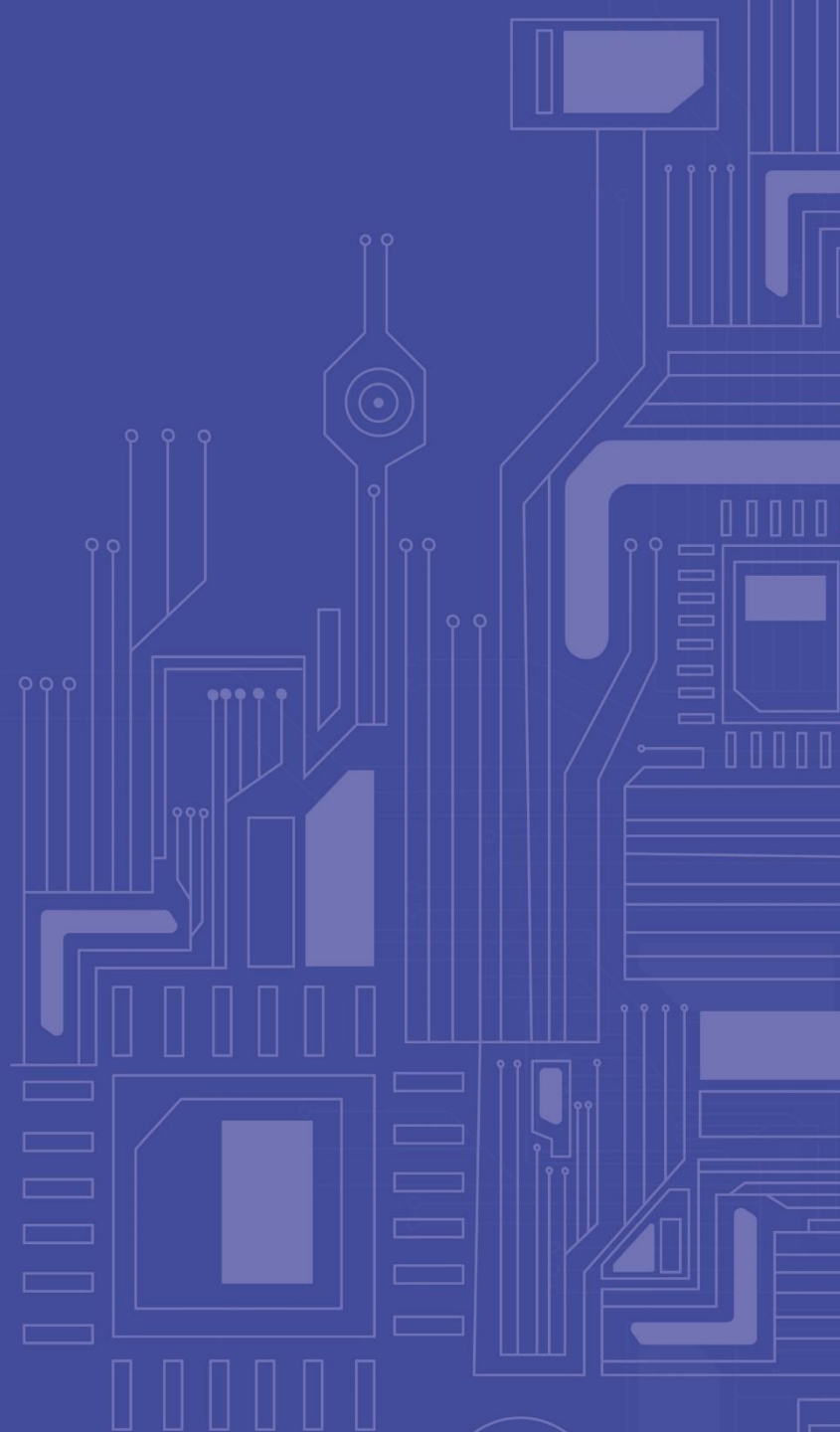
Здесь полезно обратить внимание на использование операции `collect`. Проследим за вычислением условного выражения. В нашем случае в классе `Компания` всего один объект, и он сразу становится текущим. В результате выполнения операции `self.отдел` будет получено множество объектов, соответствующих всем отделам компании. При выполнении операции `collect (служащие)` для каждого объекта-отдела по соединению с объектами класса `СЛУЖАЩИЕ` будет образовано множество объектов-служащих данного отдела, а в результате будет образовано множество объектов, соответствующих всем служащим всех отделов компании, т. е. всем служащим компании.

ПОЛУЧЕНИЕ СХЕМЫ БД ИЗ UML ДИАГРАММЫ

Рекомендация 1. Прежде чем определять в классах операции, подумайте, что вы будете делать с этими определениями в среде целевой РСУБД. Если в этой среде поддерживаются хранимые процедуры, то, возможно, некоторые операции могут быть реализованы именно с помощью такого механизма. Но если в среде РСУБД поддерживается механизм определяемых пользователями функций, возможно, он окажется более подходящим.
Рекомендация 2. Помните, что сравнительно эффективно в РСУБД реализуются только ассоциации видов «один ко многим» и «многие ко многим». Если в созданной диаграмме классов имеются ассоциации «один к одному», следует задуматься о целесообразности такого проектного решения. Реализация в среде РСУБД ассоциаций с точно заданными кратностями ролей возможна, но требует определения дополнительных триггеров, выполнение которых понизит эффективность.
Рекомендация 3. Для технологии реляционных БД агрегатные и в особенности композитные ассоциации неестественны. Подумайте о том, что вы хотите получить в реляционной БД, объявив некоторую ассоциацию агрегатной. Скорее всего, ничего.
Рекомендация 4. В спецификации UML говорится о том, что, определяя однонаправленные связи, вы можете способствовать эффективности доступа к некоторым объектам. Для технологии реляционных баз данных поддержка такого объявления вызовет дополнительные накладные расходы и тем самым снизит эффективность.
Рекомендация 5. Не злоупотребляйте возможностями OCL.
Диаграммы классов UML – это мощный инструмент для создания концептуальных схем баз данных, но, как известно, все хорошо в меру.

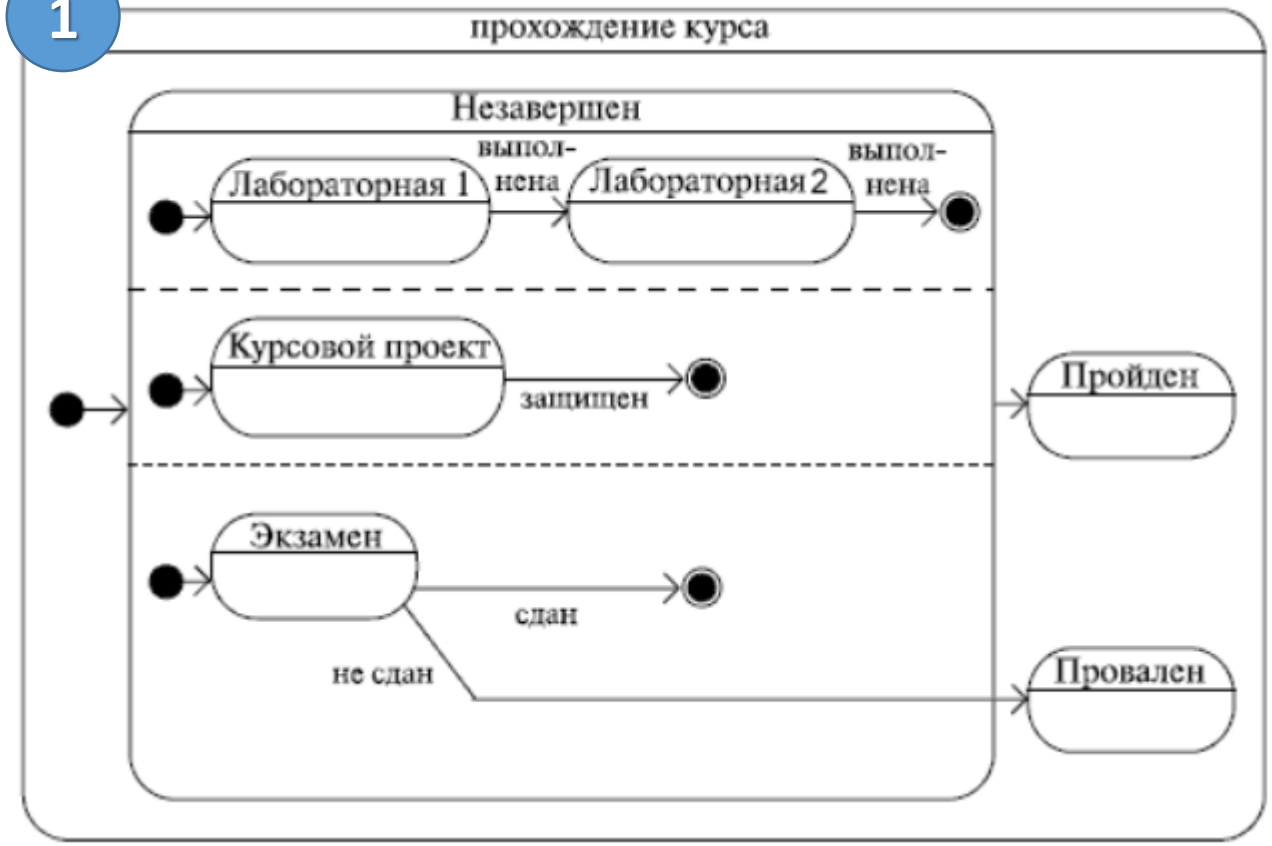

ВСЁ!

СЕМИНАР



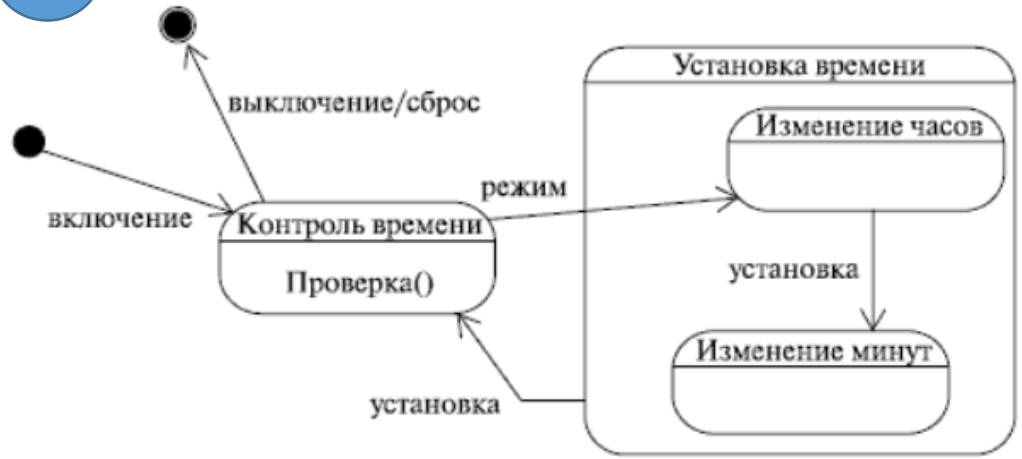
ДИАГРАММЫ СОСТОЯНИЙ

1



2

3



ДИАГРАММЫ ПОСЛЕДОВАТЕЛЬНОСТЕЙ И ВЗАИМОДЕЙСТВИЯ

Диаграмма последовательностей

Диаграмма взаимодействия

ДИАГРАММЫ АКТИВНОСТИ

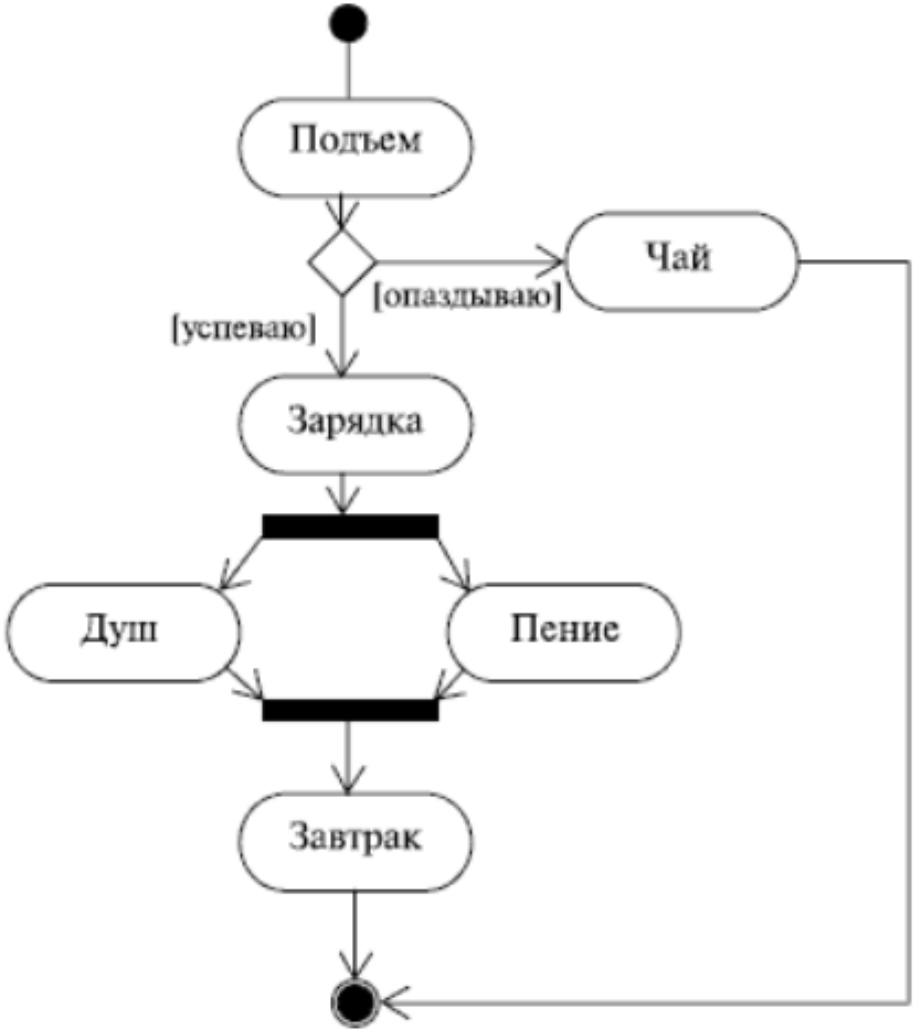
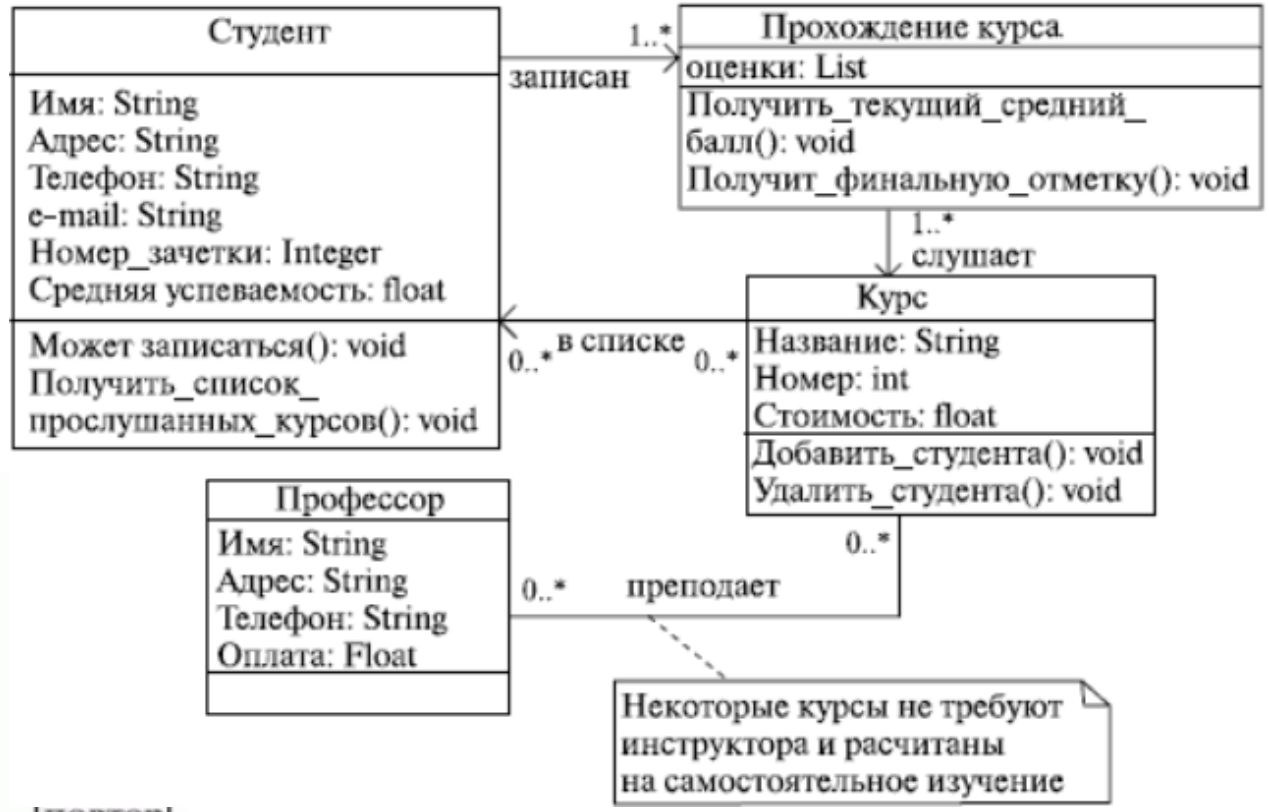
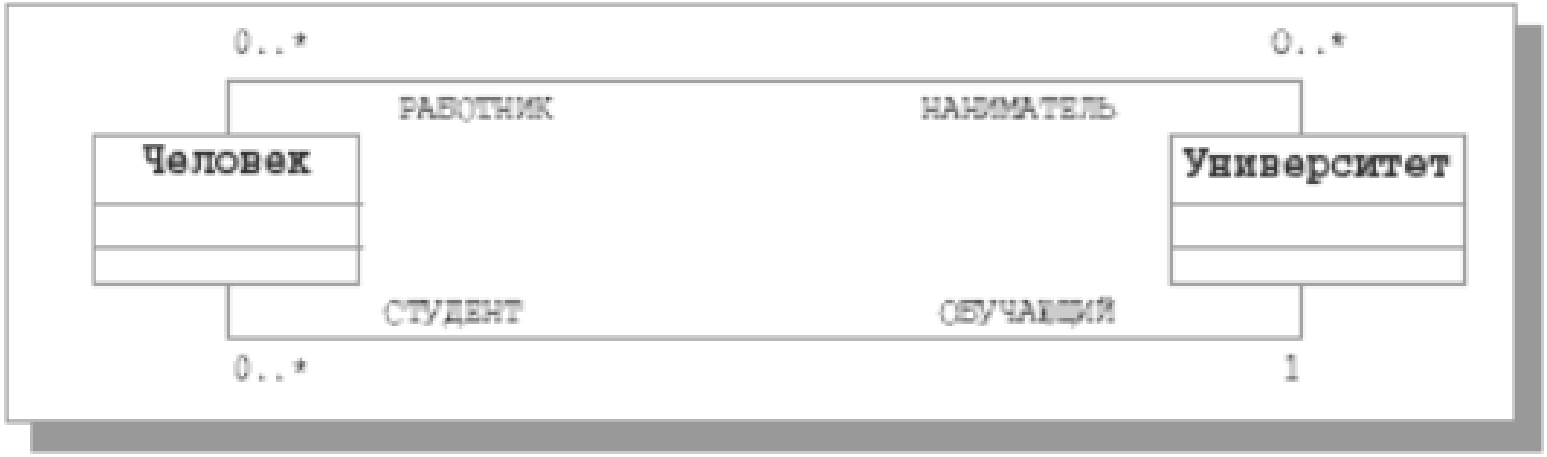


ДИАГРАММА КЛАССОВ ЯЗЫКА UML

Примеры



ПРИМЕР НА АССОЦИАЦИИ



На диаграмме классов показано, что произвольное (может быть, нулевое) число людей являются служащими произвольного числа университетов. Каждый университет обучает произвольное (может быть, нулевое) число студентов, но каждый студент может быть студентом только одного университета.