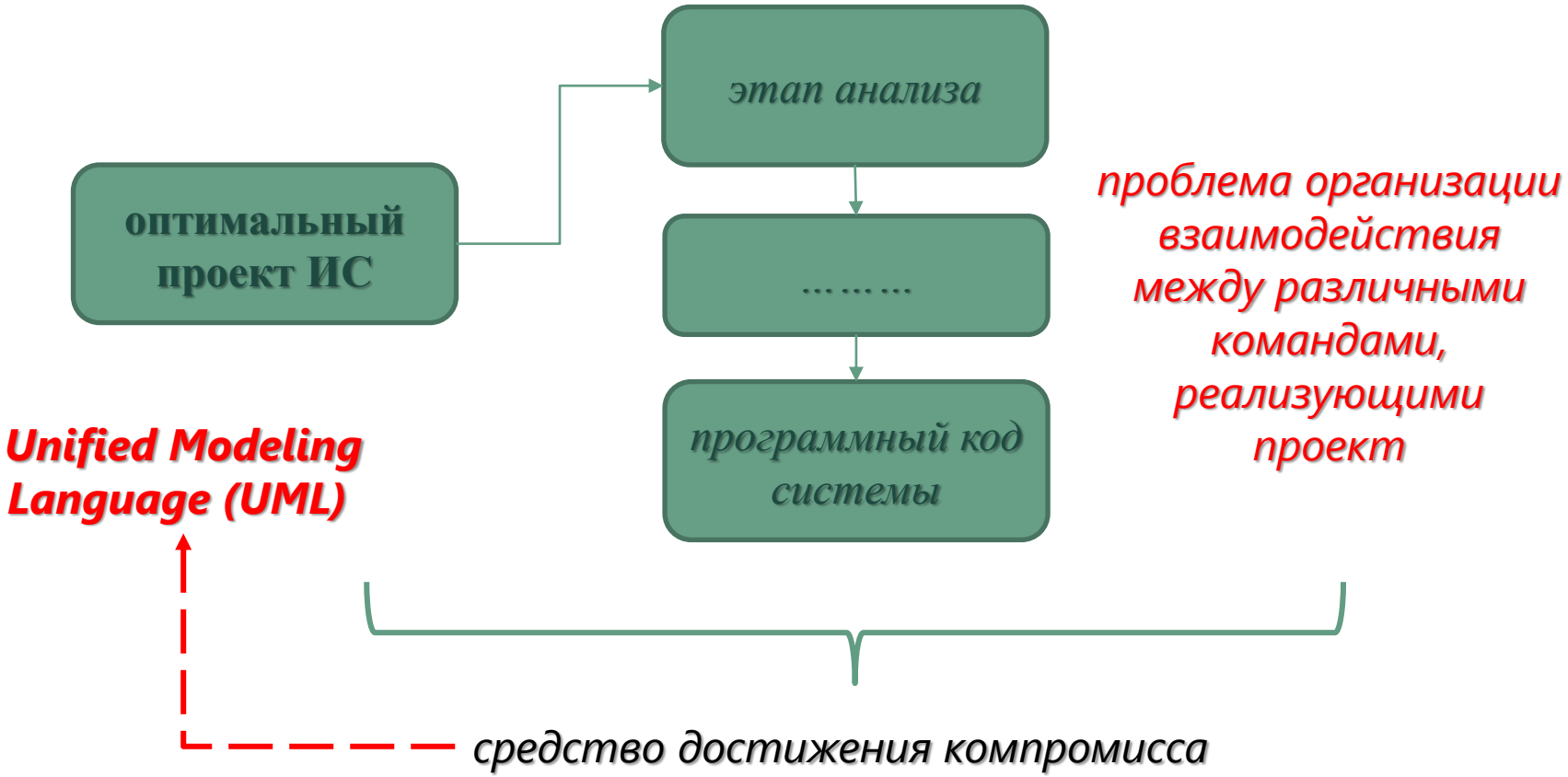
Лекция №4

Основы унифицированного языка моделирования

РТУ МИРЭА, кафедра ППИ

СОДЕРЖАНИЕ ЛЕКЦИИ:

1. Унифицированный язык объектно-ориентированного моделирования



Возникновение языка UML

Мощный толчок к появлению и развитию UML дало распространение ООП в конце 1980-х - начале 1990-х годов

- *пользователям хотелось получить единый язык моделирования, который объединил бы в себе всю мощь объектно-ориентированного подхода и давал бы четкую модель системы, отражающую все ее значимые стороны.*

К середине девяностых явными лидерами в этой области стали методы:

- Booch (Grady Booch)
- OMT-2 (Jim Rumbaugh)
- OOSE (Ivar Jacobson)

создание *UML* началось в октябре 1994 г., когда Джим Рамбо и Гради Буч из Rational *Software* Corporation стали работать над объединением своих методов *OMT* и Booch.

осенью 1995 г. увидела свет первая черновая версия объединенной методологии, которую они назвали Unified Method 0.8.

После присоединения в конце 1995 г. к Rational Software Corporation Айвара Якобсона и его фирмы Objectory, усилия трех создателей наиболее распространенных объектно-ориентированных методологий были объединены и направлены на созданиеUML.

Основные характеристики языка UML

Характеристики UML

- *является языком визуального моделирования, который обеспечивает разработку репрезентативных моделей для организации взаимодействия заказчика и разработчика ИС, различных групп разработчиков ИС;*
- *содержит механизмы расширения и специализации базовых концепций языка.*

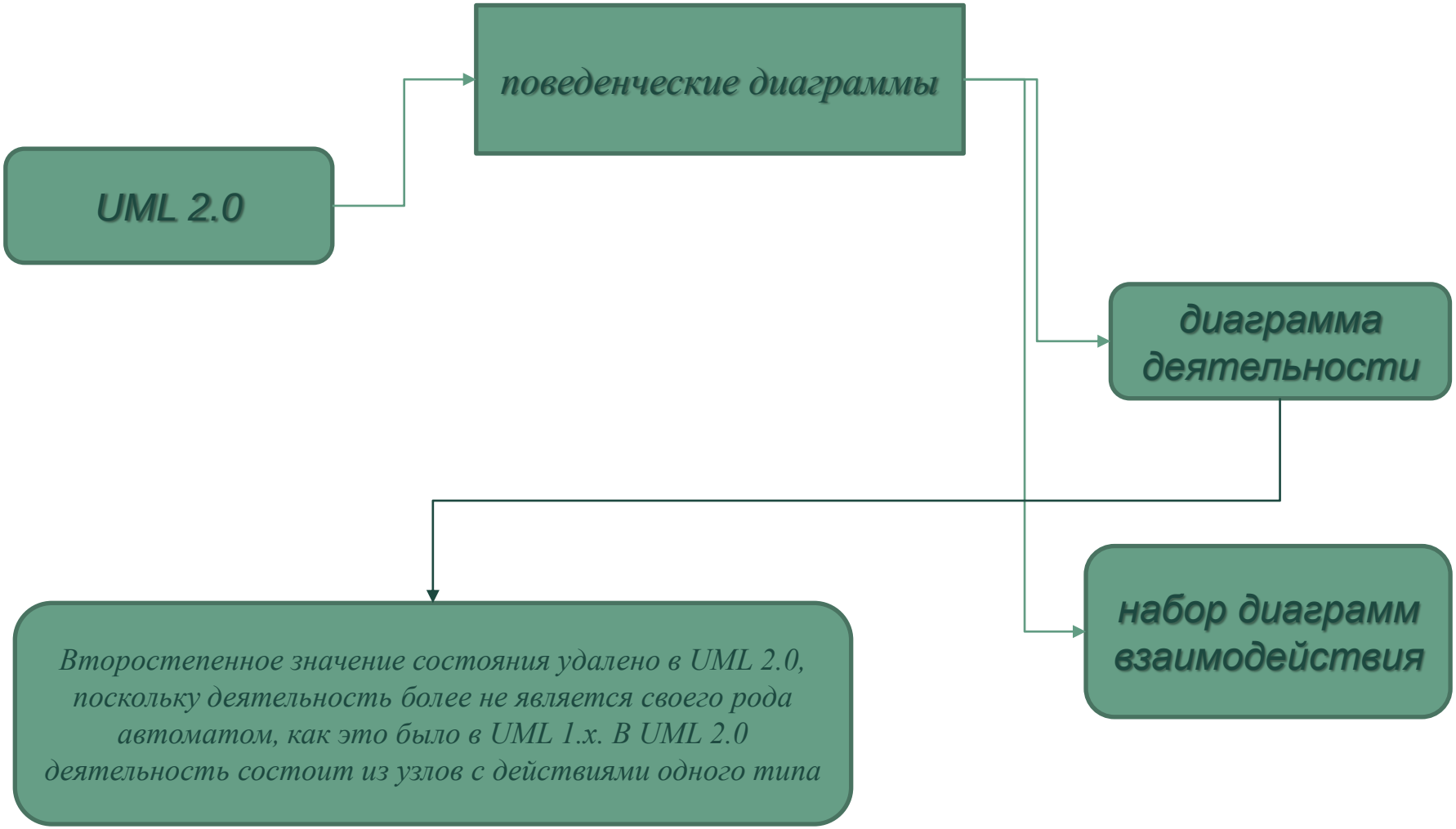
Возможности UML

- *строить модели на основе средств ядра, без использования механизмов расширения для большинства типовых приложений;*
- *добавлять при необходимости новые элементы и условные обозначения, если они не входят в ядро, или специализировать компоненты, систему условных обозначений (нотацию) и ограничения для конкретных предметных областей.*

Версии UML

Версия	Дата принятия
1.1	ноябрь 1997
1.3	март 2000
1.4	сентябрь 2001
1.4.2	июль 2004
1.5	март 2003
2.0	июль 2005
2.1	формально не была принята
2.1.1	август 2007
2.1.2	ноябрь 2007
2.2	февраль 2009
2.3	май 2010
2.4 beta 2	март 2011
2.5	июнь 2015
2.5.1	декабрь 2017

Сравнительные характеристики UML 1.x и UML 2.0

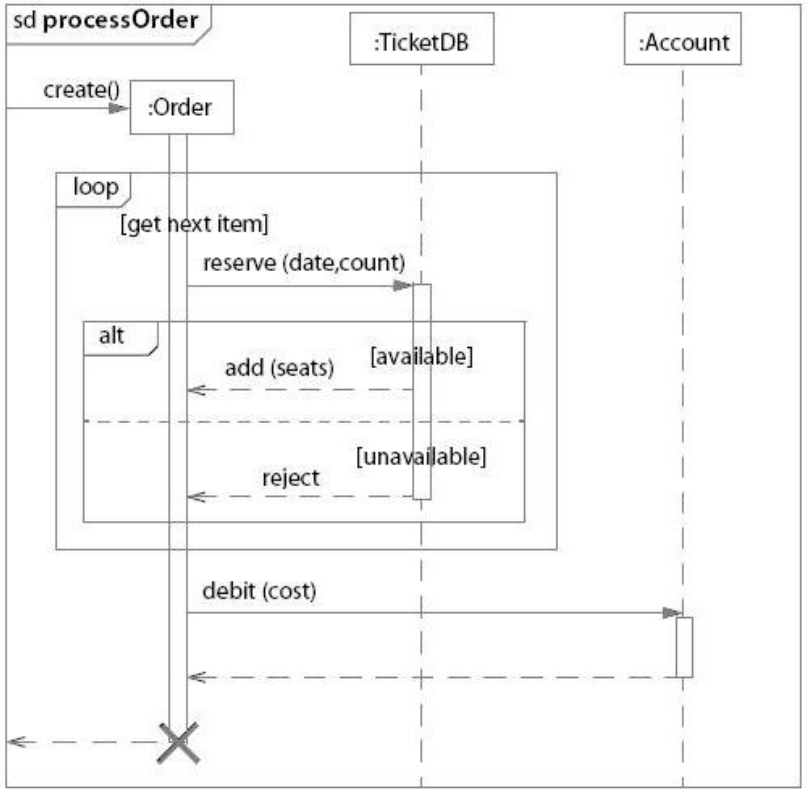


Новые возможности для диаграммы последовательности в UML2

UML 1.x:	*UML 2.x:*
• *представление цикла - примечание, присоединенное к сообщению с указанием цикла (выполнялось до тех пока условие цикла было верно);* • *выполнение экземпляра показывает текущий управляющий элемент, который исполняется объектом в некоторый момент времени при получении сообщения.*	• *для циклов существует специальное представление - фрагмент, объединяющий части диаграммы, моделирующие различные потоки;* • *использование общих участков для нескольких взаимодействий: экземпляры взаимодействия допускают расщепления взаимодействий на многоразовые;* • *объекты создаются и уничтожаются.*

Формы диаграмм последовательности в UML 1.x и UML 2.x

В зависимости от количества сценариев в рамках одного прецедента, диаграммы последовательностей можно использовать в двух формах:



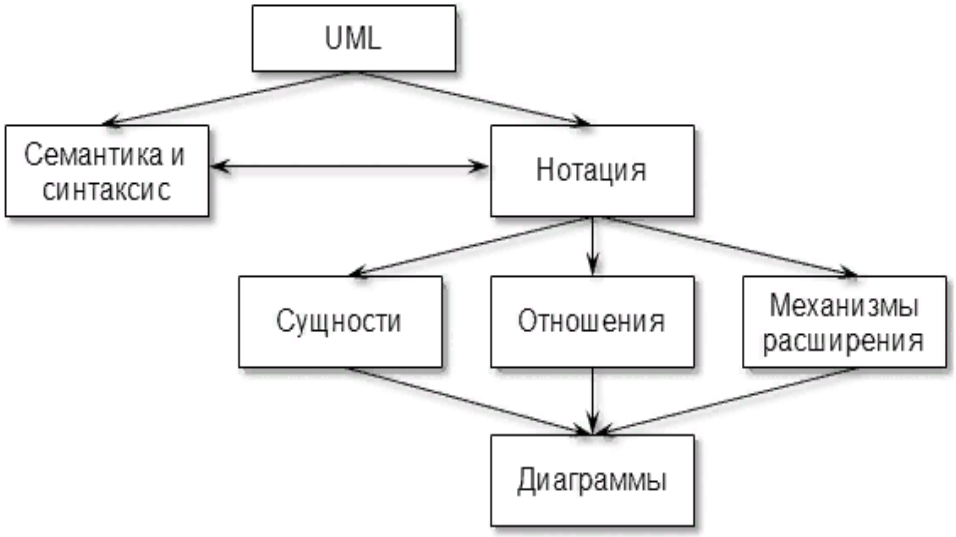
*UML 1.x - **отдельная форма (**для представления одного сценария варианта):*

- *подробно описывает определенный сценарий, документируя одно возможное **взаимодействие без условий, ветвей или циклов;***
- *различные сценарии одного и того же прецедента представляются на различных диаграммах последовательностей.*

*UML 2.x - **общая форма (**для представления нескольких сценариев варианта)**:***

- *описывает все сценарии прецедента, **используя условия, ветви и циклы;***
- *возможно использование как **отдельной формы, так и общей формы.***

2. Синтаксис и семантика основных объектов UML



- **синтаксис** (syntax), то есть определение правил составления конструкций языка;
- **семантика** (semantics), то есть определение правил приписывания смысла конструкциям языка;
- **прагматика** (pragmatics), то есть определение правил использования конструкций языка для достижения определенных целей.

Одним из назначений UML является:

- создание таких моделей, для которых возможна автоматическая генерация программного кода (или фрагментов кода) соответствующих приложений;
- автоматическое построение модели по коду готового приложения.

Почему язык UML называют унифицированным?

Жизненный цикл разработки:

- UML предоставляет визуальный синтаксис для моделирования на протяжении всего ЖЦ разработки ПО – от постановки требований до реализации.

Области приложений:

- UML используется для моделирования всех аспектов – от аппаратных встроенных систем реального времени до систем поддержки принятия решений.

Языки реализации и платформы:

- UML является независимым от языков и платформ.

Процессы разработки:

- хотя UP и его разновидности, вероятно, являются предпочтительными процессами разработки ОО систем, UML может поддерживать (и поддерживает) множество других процессов разработки ПО.

Собственные внутренние концепции:

- UML поистине стойко стремится сохранить последовательность и постоянство применения небольшого набора своих внутренних концепций.

Унифицированный процесс и UML

Важно понимать, что UML не предлагает нам какой-либо методологии моделирования.

UML предоставляет собой лишь визуальный синтаксис, который можно использовать для создания моделей.

Он используется в различных методологиях, но наиболее адаптированная под него методология называется UP.

*Унифицированный процесс (**Unified Process, UP**) – это методология.*

Она указывает на исполнителей, действия и артефакты, которые необходимо использовать, осуществить или создать для моделирования программной системы.

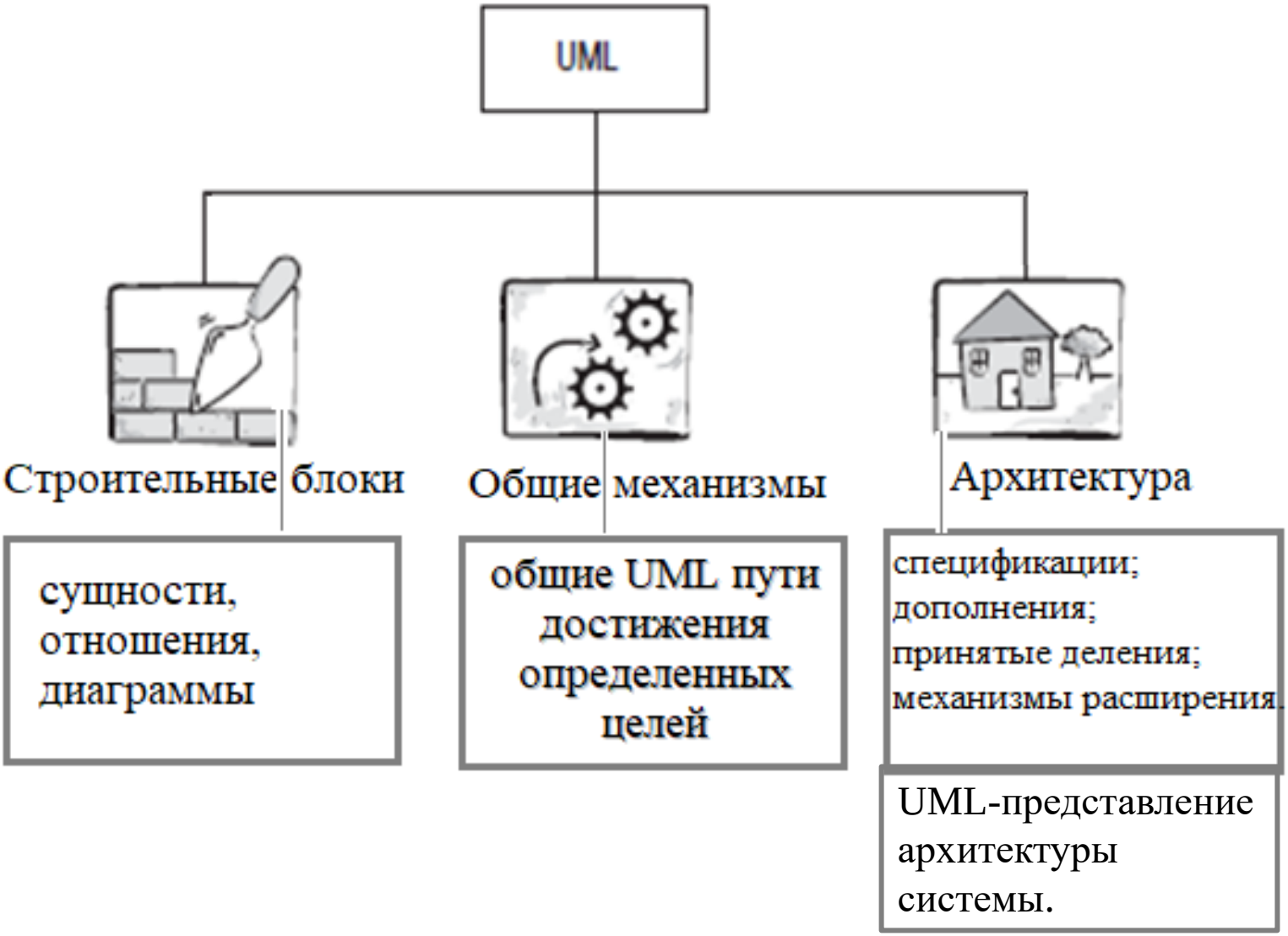
***UML** не привязан к какой- либо конкретной методологии или жизненному циклу.*

На самом деле он может использоваться со всеми существующими методологиями.

***UP** использует UML в качестве базового синтаксиса визуального моделирования.*

*Основная идея **UML** – возможность моделировать программное обеспечение и другие системы как наборы взаимодействующих объектов.*

Структура UML



Аспекты UML модели

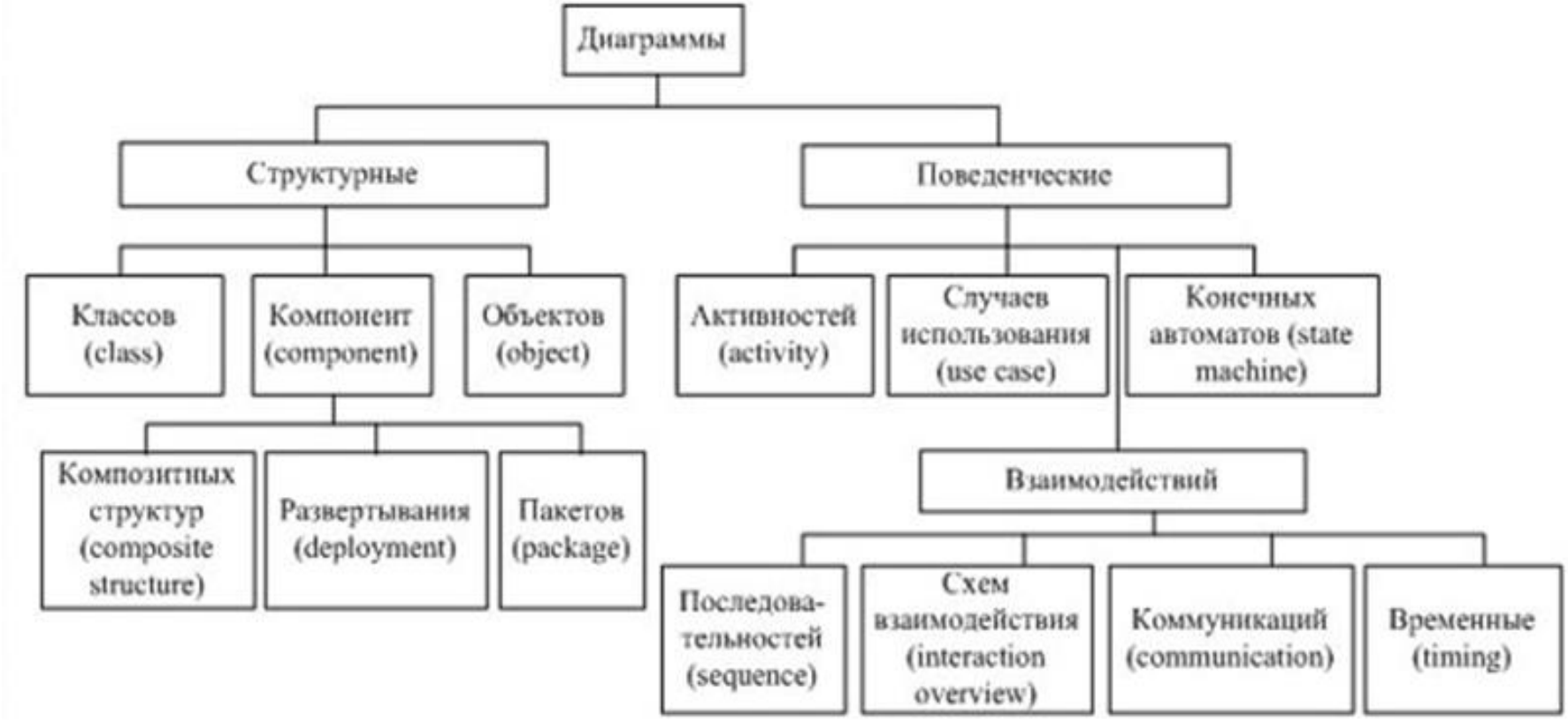
Статическая структура

- *– описывает, какие типы объектов важны для моделирования системы и как они взаимосвязаны.*

Динамическое поведение

- *– описывает жизненные циклы этих объектов и то, как они взаимодействуют друг с другом для обеспечения требуемой функциональности системы.*

Иерархия типов диаграмм



Основные понятия UML

1. Объект (object) — сущность, обладающая уникальностью и инкапсулирующая в себе состояние и поведение.

2. Класс (class) — описание множества объектов с общими атрибутами, определяющими состояние, и операциями, определяющими поведение.

3. Интерфейс (interface) — именованное множество операций, определяющее набор услуг, которые могут быть запрошены потребителем и предоставлены поставщиком услуг.

4. Кооперация (collaboration) — совокупность объектов, которые взаимодействуют для достижения некоторой цели.

5. Действующее лицо (actor) — сущность, находящаяся вне моделируемой системы и непосредственно взаимодействующая с ней.

6. Компонент (component) — модульная часть системы с четко определенным набором требуемых и предоставляемых интерфейсов.

7. Артефакт (artifact) — элемент информации, который используется или порождается в процессе разработки программного обеспечения.

8. Узел (node) — вычислительный ресурс, на котором размещаются и при необходимости выполняются артефакты.

9. Состояние (state) — период в жизненном цикле объекта, находясь в котором объект удовлетворяет некоторому условию и осуществляет собственную деятельность или ожидает наступления некоторого события.

10. Деятельность (activity) можно считать частным случаем состояния, который характеризуется продолжительными (по времени) не атомарными вычислениями.

11. Действие (action) — примитивное атомарное вычисление.

12. Вариант использования (use case) — множество сценариев, объединенных по некоторому критерию и описывающих последовательности производимых системой действий, доставляющих значимый для некоторого действующего лица результат.

13. Пакет (package) — группа элементов модели (в том числе пакетов).

Гибкость UML

Сокращенные модели *– некоторые элементы присутствуют в заднем плане, но скрыты в той или иной диаграмме для упрощения представления*

Неполные модели *– некоторые элементы модели могут быть полностью пропущены*

Несогласованные модели *–может содержать противоречия*

3. Понятие модели и диаграмм

Диаграмма

графическое представление множества элементов

с помощью диаграмм можно визуализировать систему с различных точек зрения

ни одна отдельная диаграмма не является моделью, лишь набор диаграмм составляет модель системы

Два измерения моделей

Графическое измерение

- *позволяющее визуализировать модель с помощью диаграмм и пиктограмм*

Текстовое измерение

- *состоящее из спецификаций (текстового описания семантики элементов) различных элементов модели.*

Диаграммы и модели

Диаграммы

- - только визуальные представления или картины модели (проекции заднего семантического плана)
- *сущность или отношение могут быть удалены с диаграммы, но они по-прежнему продолжают существовать в модели.*

Модели

- - текстовое представление (описание) наиболее существенных характеристик оригинала, состоящее из спецификаций, которое может быть дополнено визуальным представлением
- *новые сущности или новые отношения при создании добавляются в модель*

Диаграмма классов

РТУ МИРЭ, кафедра ППИ

Диаграмма классов

Диаграмма классов (*class diagram*) — структурная диаграмма языка моделирования UML, демонстрирующая общую структуру иерархии классов системы, их кооперацией, атрибутов, методов, интерфейсов и взаимосвязей между ними.

Диаграмма классов применяется для:

- документирования;
- визуализации;
- конструирования (посредством прямого или обратного проектирования).

Класс (class)- *категория вещей, которые имеют общие атрибуты и операции.*

Уровни диаграмм классов

Аналитический уровень

- класс содержит в себе только набросок общих контуров системы и работает как логическая концепция предметной области или программного продукта **(сущности предметной области)**

Уровень проектирования

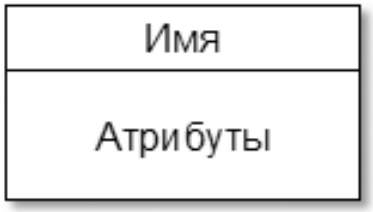
- отражает основные проектные решения касательно распределения информации и планируемой функциональности, объединяя в себе сведения о состоянии и операциях **(элементы программной системы)**

Уровень реализации

- дорабатывается до такого вида, в каком он максимально удобен для воплощения в выбранной среде разработки;
- при этом не воспрещается опустить в нём те общие свойства, которые не применяются на выбранном языке программирования **(элементы программной системы)**

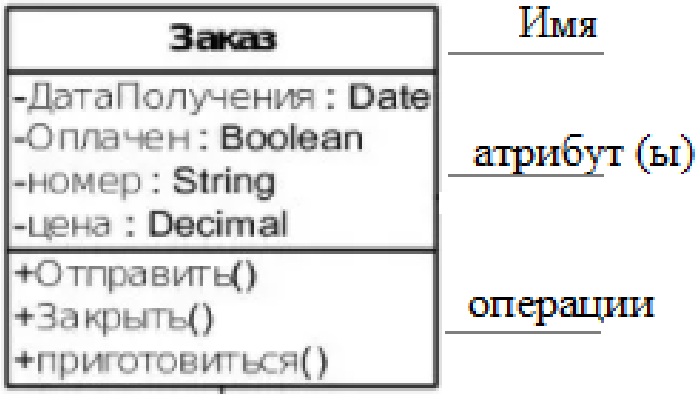
Основные элементы диаграммы классов

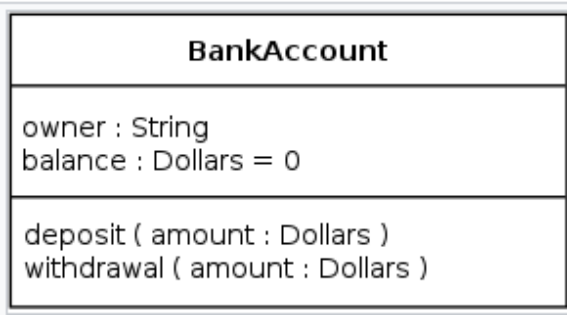
- Классы (имя, атрибуты, операции)
- отношения



Классы

- Изображаются в рамках , содержащих три компонента - горизонтально вытянутый прямоугольник (с одним, двумя, тремя разделами/секциями)





В верхней части написано имя класса – секция имени

- выравнивается по центру и пишется полужирным шрифтом;
- начинается с заглавной буквы;
- если класс абстрактный — то его имя пишется полужирным курсивом.

Посередине располагаются поля (атрибуты) класса –секция атрибутов

- выровнены по левому краю и начинаются с маленькой буквы;

Нижняя часть содержит операции (методы) класса – секция методов

- выровнены по левому краю и пишутся с маленькой буквы.

Секция «имя»

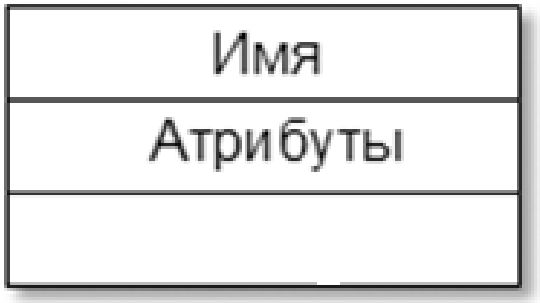
Обязательным элементом обозначения класса на диаграмме является его **имя**. Оно должно быть уникальным в пределах пакета. Если класс является абстрактным, то его имя пишется курсивом.

- **Абстрактный класс** – это класс, на основе которого нельзя создать объекты. Такие классы используются в качестве шаблона для дочерних классов при наследовании.

- В секции имени класса может быть указан стереотип (например, **«entity», «boundary», «interface», «control»** и т. п.).

Секция «атрибуты»

Во второй секции
каждому ***атрибуту*** соответствует отдельная строка со следующей спецификацией:
[видимость] [/] имя [: тип [‘[‘кратность‘]‘] [= исходное значение]] [‘{‘модификаторы’}’].

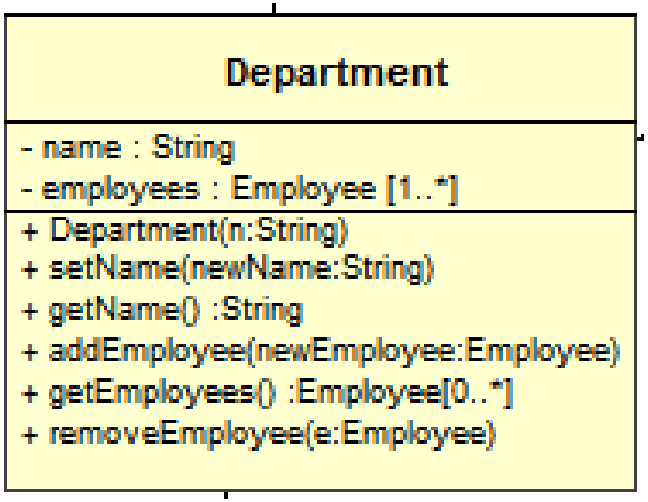


- Атрибуты могут быть одинаковыми в разных классах. Например атрибут ФИО в классе *преподаватель* подразумевает ФИО преподавателя, а в классе студент ФИО студента. Но одинаковые названия атрибутов внутри одного класса не могут быть.

Секция атрибуты

- Атрибуты содержат огромный набор спецификаций (набор параметров, которыми характеризуются сами атрибуты) .
- Для атрибута обязательным является наличие **имени**, все остальные спецификации являются дополнительными, то есть вариативными (например, видимость, производность, кратность, тип данных и т.д.) Они могут не поместиться на диаграмме, но они есть на семантическом заднем плане и они выполняются
- **Атрибуты [видимость] [/] имя [‘[‘кратность’]’] =исходное значение]] [‘{‘модификаторы’}’]**

Квадратные скобки означают, что что элемент спецификации может отсутствовать.



Видимость

Видимость отображается с помощью следующих символов:
- "+" – общедоступный атрибут (англ. public) – доступен для чтения и модификации из объектов любого класса;
- "#" – защищенный атрибут (англ. protected) – доступен только объектам описываемого класса и его потомкам при наследовании;
- "–" – закрытый атрибут (англ. private) – доступен только объектам описываемого класса;
- "~" – пакетный атрибут (англ. package) – доступен только объектам классов, входящих в тот же пакет.

+	Публичный (Public)
-	Приватный (Private)
#	Защищённый (Protected)
~	Пакет (Package)

Видимость *(англ. visibility) характеризует возможность чтения и модификации значения атрибута объекта описываемого класса, из объектов других классов. Модификация значения возможна лишь при условии, что атрибут не является константой.*

Производность

- Символ «/» перед именем атрибута указывает на то, что он является производным (т.е. его значение вычисляется из значений других атрибутов или ассоциаций).
- Формально производный элемент излишен, он вводится в модель для ясности или наглядности.

Имя (name) атрибута

- **Имя атрибута** представляет собой строку текста, которая используется для его идентификации. Оно должно быть уникальным в пределах класса.
- Имена атрибуты могут быть одинаковыми только в разных классах. Например атрибут с именем ФИО в классе «преподаватель» подразумевает ФИО преподавателя, а в классе «студент» ФИО студента. Но одинаковые имена атрибутов внутри одного класса не могут быть!!
- Желательно уточнять одинаковые имена атрибутов даже в разных классах, чтобы максимально точно идентифицировать назначение данного атрибута.

Тип (type) атрибута

- выбирается исходя из семантики значений, которые должны храниться в атрибуте, и, как правило, возможностей целевого языка программирования по представлению этих значений.
- тип соответствует одному из стандартных типов, определенных в этом языке (например, String, Boolean, Integer, Color и т. д.) или имени класса, на объект которого в этом атрибуте будет храниться ссылка (в этом случае класс, имя которого указано в качестве типа, должен быть определен на диаграмме или в модели).

Кратность

- **Кратность (multiplicity)** атрибута характеризует количество значений, которые можно хранить в атрибуте. Если кратность атрибута не указана, то по умолчанию принимается ее значение, равное 1, т. е. атрибут является атомарным. Такой вариант допускает и отсутствие значения в атрибуте (null).
- Для атрибута, представляющего собой массив, множество, список и т. п. , требуется указание кратности, которая записывается после типа в квадратных скобках.

Варианты указания кратности, имеющие смысл, могут быть следующие:

- - [0. . *] или [*] – количество хранимых значений может принимать любое положительное целое число, большее или равное 0. Данный вариант применяется при описании массивов фиксированного размера. При этом не обязательно, чтобы все элементы массива имели конкретные значения;
- - [0. . <число>] – применяется при описании двумерных массивов. Аналогичным образом можно описать трехмерные, четырехмерные и т. д. массивы.

Модификатор (modifier)

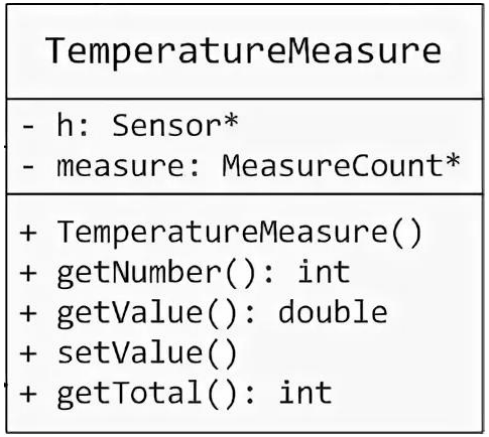
Модификатор (modifier) описывает особенности реализации атрибута, например:

- - {final} / {read. Only} – атрибут является константой, т. е. доступен только для чтения;
- - {static} – атрибут при выполнении программы в конкретный момент времени будет иметь одно и то же значение для всех объектов класса;
- - {transient} – атрибут и его значение при записи объекта в БД или файл (сериализации объекта) не должны запоминаться;
- - {redefines <имя атрибута родительского класса>} – атрибут переопределяет (заменяет) атрибут родительского класса;
- - {id} – значение атрибута используется в качестве идентификатора объекта класса;
- - {seq} / {sequence} – значения неатомарного атрибута хранятся упорядочено (к ним можно обращаться по индексу или выполнять перебор в соответствии с порядком их добавления в список/массив/множество) и могут повторяться.

Секция операции (методы)

[видимость] имя ([список параметров]) [: тип] [‘{‘свойства'}'].

- Имя и кратность параметра задаются по тем же правилам, что и для атрибутов класса. Тип параметра – тип значений, которые может принимать параметр. Значение по умолчанию – значение, которое передается в метод, если при вызове метода данный параметр не определен.
- Тип метода – тип результата, возвращаемого методом. Если тип не указан, то метод не возвращает никакого результата (в языках программирования такие методы, как правило, обозначаются модификатором void).



Свойства

Свойства служат для указания специфических свойств метода, например:

- - {native} – реализация метода зависит от платформы (операционной системы);
- - {abstract} – метод в описываемом классе не имеет тела. Код метода должен быть определен в дочерних классах;
- - {sequential} – метод допускает только последовательный вызов. Парралельный вызов операции может вызвать сбой программы;
- - {guarded} – метод автоматически блокируется (ждет очереди) до завершения других вызовов (экземпляров) метода;
- - {concurrent} – допускается параллельное (одновременное) выполнение нескольких вызовов (экземпляров) метода;
- - {query} – метод не меняет состояние системы. Как правило, в языках программирования имена таких методов начинаются на get или is;
- - {redefines <имя метода родительского класса>} – метод переопределяет (заменяет) метод родительского класса.

Описание операций (методов)

конструктор

- – метод, создающий и инициализирующий объект. В Java имя конструктора совпадает с именем класса;

деструктор

- – метод, уничтожающий объект. В некоторых языках программирования (в частности в Java) определение деструкторов не требуется, так как очистка памяти от неиспользуемых объектов (сборка мусора) выполняется автоматически;

модификатор

- – метод, который изменяет состояние объекта (значения атрибутов). Имена модификаторов начинаются, как правило, со слова set (англ. – установить). Например, установить атрибуту Name новое значение set. Name(new. Name : String);

селектор

- – метод, который может только считывать значения атрибутов объекта, но не изменяет их. Имена селекторов начинаются, как правило, со слов get (англ. – получить) или is при возврате логического результата. Например, считать значение атрибута Name – get. Name() или определить видимость на экране элемента графического интерфейса – is. Visible();

итератор

- – метод, позволяющий организовать доступ к элементам объекта. Например, для объекта, представляющего собой множество Set или список List, это могут быть методы перехода к первому элементу first(), следующему next(), предыдущему previous() и т. п. ;

событие

- – метод, запускаемый на выполнение автоматически при соблюдении определенных условий.

Примеры указания методов

Спецификация метода в UML

- Генерируемый код для языка Java "constructor" + Text. Field. Int(value : public Text. Field. Int(int value, int, length : int, alligment : int, length, int alligment, Font font. Field) font. Field : Font) { } + save. Data() public void save. Data() {return; } + is. Visible() : boolean public boolean is. Visible() {return false; } # init(text : String, icon : Icon) protected void init(String text, Icon icon) {return; }

Области действия диаграммы классов

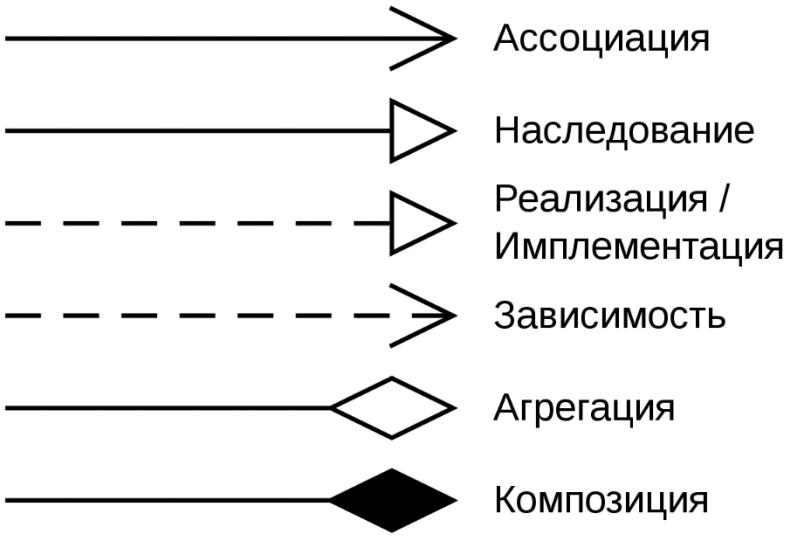
Члены классификаторы (static)

- Область действия — сам класс.
  - Значения полей одинаковы для всех экземпляров в данной единице трансляции
  - Вызов метода не меняет состояние объекта
- Для указания принадлежности к классификатору, имя подчёркивается

Члены экземпляры

- Область действия — объект.
  - Значения полей могут отличаться в разных объектах
  - Методы могут изменять поля

Взаимосвязи объектов классов



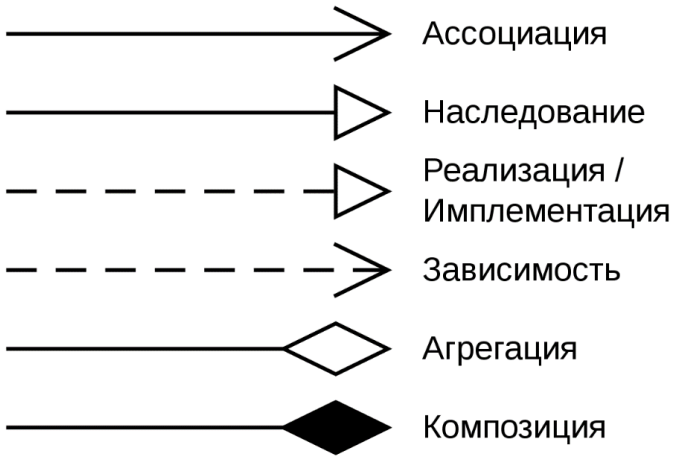
Ассоциация может быть именованной, и на концах представляющей её линии могут быть подписаны роли, принадлежности, индикаторы, мультипликаторы, видимости или другие свойства.

Ассоциация показывает, что объекты одной сущности (класса) связаны с объектами другой сущности таким образом, что можно перемещаться от объектов одного класса к другому. Является общим случаем композиции и агрегации.

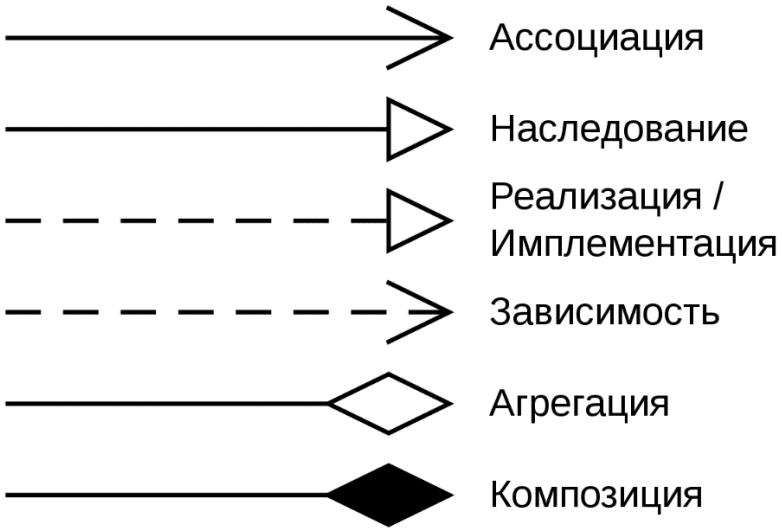
Двойные ассоциации представляются линией без стрелочек на концах, соединяющей два классовых блока.

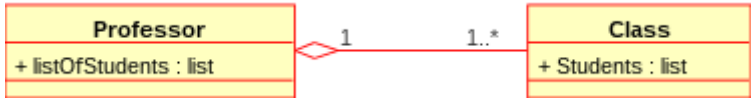
Зависимость

- ***Зависимость*** обозначает такое отношение между классами, что изменение спецификации класса-поставщика может повлиять на работу зависимого класса, но не наоборот.



Агрегация

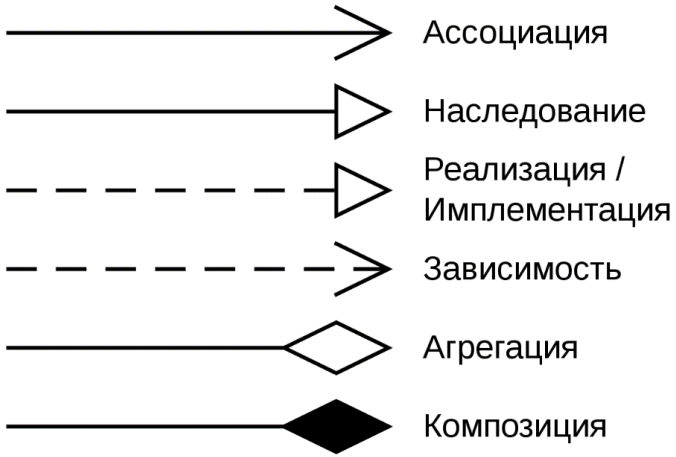




Агрегация — это разновидность ассоциации при отношении между целым и его частями (может быть именованной).
Агрегация встречается, когда один класс является коллекцией или контейнером других.
Причём по умолчанию, агрегацией называют *агрегацию по ссылке*, то есть когда время существования содержащихся классов не зависит от времени существования содержащего их класса. Если контейнер будет уничтожен, то его содержимое — нет.
Графически агрегация представляется пустым ромбом на блоке класса, и линией, идущей от этого ромба к содержащемуся классу.

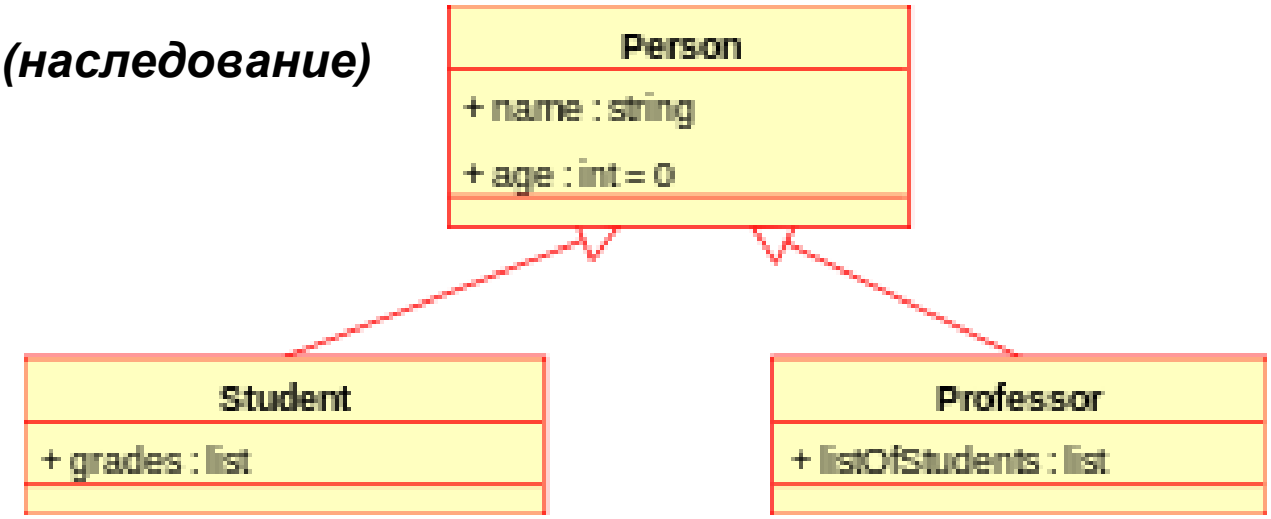
Композиция

- ***Композиция*** — более строгий вариант агрегации. Известна также как агрегация по значению.
- *Композиция* имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.
- Графически представляется, как и агрегация, но с закрашенным ромбиком.



Взаимосвязи классов

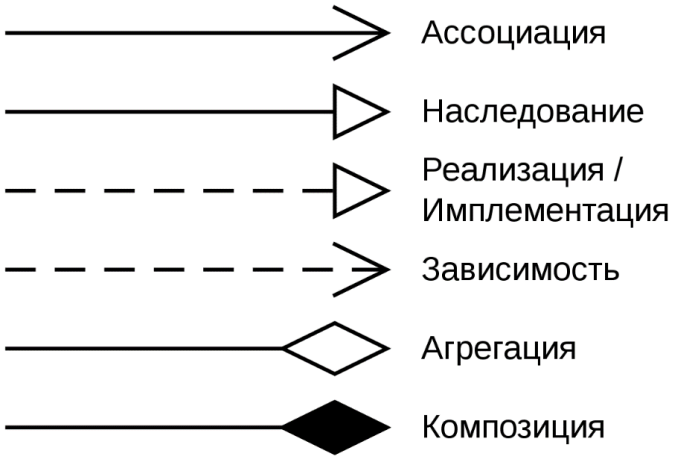
- ***Обобщение (наследование)***



Обобщение (Generalization) показывает, что один из двух связанных классов (*подтип*) является частной формой другого (*надтипа*), который называется **обобщением** первого. На практике это означает, что любой экземпляр подтипа является также экземпляром надтипа. Например: животные — супертип млекопитающих, которые, в свою очередь, — супертип приматов, и так далее. Эта взаимосвязь легче всего описывается фразой «А — это Б» (приматы — это млекопитающие, млекопитающие — это животные).// Графически обобщение представляется линией с пустым треугольником у супертипа.
Обобщение также известно как наследование или «is a» взаимосвязь (или отношение «является»).

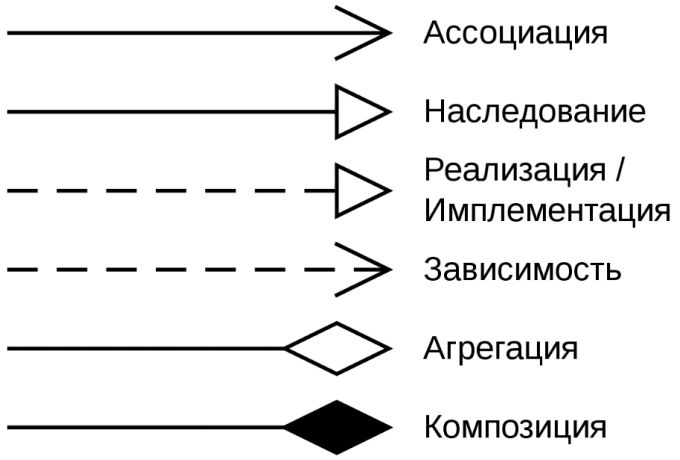
Взаимосвязи классов

- ***Реализация*** — отношение между двумя элементами модели, в котором один элемент (*клиент*) реализует поведение, заданное другим (*поставщиком*). Реализация — отношение целое-часть. Графически реализация представляется так же, как и наследование, но с пунктирной линией.
- Поставщик, как правило, является абстрактным классом или классом-интерфейсом.



Общая взаимосвязь

- ***Зависимость (dependency)*** — это слабая форма отношения использования, при которой изменение в спецификации одного влечёт за собой изменение другого, причём обратное не обязательно. Возникает, когда объект выступает, например, в форме параметра или локальной переменной.
- Графически представляется штриховой стрелкой, идущей от зависимого элемента к тому, от которого он зависит.
- Существует несколько именованных вариантов.
- Зависимость может быть между экземплярами, классами или экземпляром и классом.

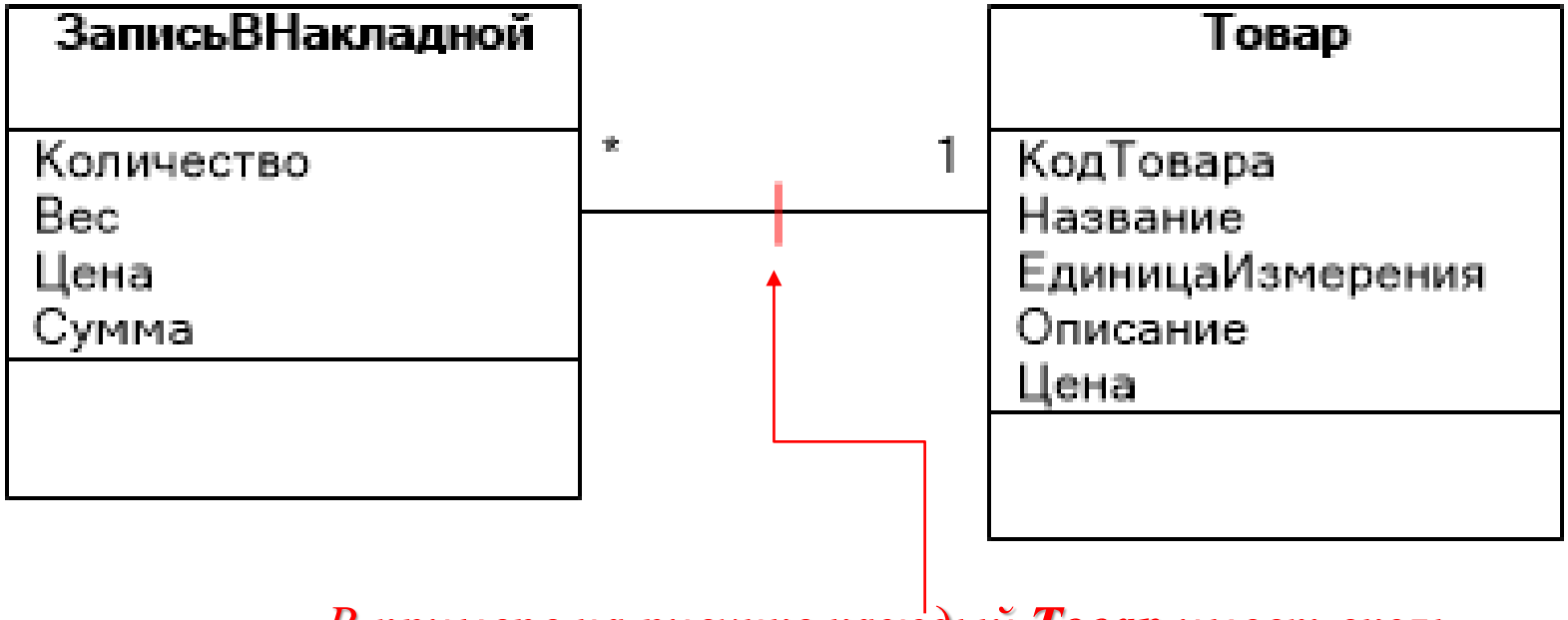


Мощность отношений (Кратность)

Мощность отношения (мультипликатор) означает число связей между каждым экземпляром класса (объектом) в начале линии с экземпляром класса в её конце

нотация	объяснение	пример
0..1	Ноль или один экземпляр	Кошка имеет хозяина.
1	Обязательно один экземпляр	у кошки одна мать
0..* или *	Ноль или более экземпляров	у кошки могут быть, а может и не быть котят
1..*	Один или более экземпляров	у кошки есть хотя бы одно место, где она спит

Пример кратности



*В примере на рисунке каждый **Товар** имеет сколь угодно **Записей** в накладной, но каждая **Запись** в накладной обязательно один **Товар**.*

Правила разработки диаграмм классов

1. За основу диаграммы классов при ее разработке берется диаграмма классов анализа.

2. Для классов должны быть определены и специфицированы все атрибуты и методы. Их спецификация, как правило, выполняется с учетом выбранного языка программирования.

3. При определении методов рекомендуется использовать сообщения с ранее разработанных диаграмм последовательности и коммуникации.

4. Детальное проектирование **граничных классов,** как правило, не требуется. При проектировании таких классов основное внимание следует уделять особенностям отображения информации и специфичным операциям, которые возникают при диалоге пользователя с системой. Граничные классы, определяющие интерфейс взаимодействия с другими системами, требуют детального проектирования.

5. Для проектирования ***классов-сущностей*** можно применять подходы, используемые при проектировании БД, особенно в том случае, если данные будут храниться в таблицах БД. Если представление данных в БД и классах отличается друг от друга и в качестве хранилища информации будет применяться реляционная база данных, то рекомендуется разработать отдельную диаграмму классов, описывающую состав и структуру БД.

6. Несмотря на то, что каждому объекту при выполнении программы автоматически назначается уникальный идентификатор, рекомендуется для классов-сущностей явно определять атрибуты, хранящие значения первичного ключа.

7. В отличие от реляционных БД поощряется использование в классах многозначных атрибутов в виде массивов, множеств, списков и т. д.

8. ***Управляющие классы*** следует проектировать только в случаях крайней необходимости – управления сложным взаимодействием объектов, реализации сложной бизнес-логики и вычислений, контроля целостности объектов и т. п. В противном случае функциональность этого класса лучше распределить между соответствующими граничными классами и классами-сущностями.

Спасибо за внимание

РТУ МИРЭ, кафедра ППИ