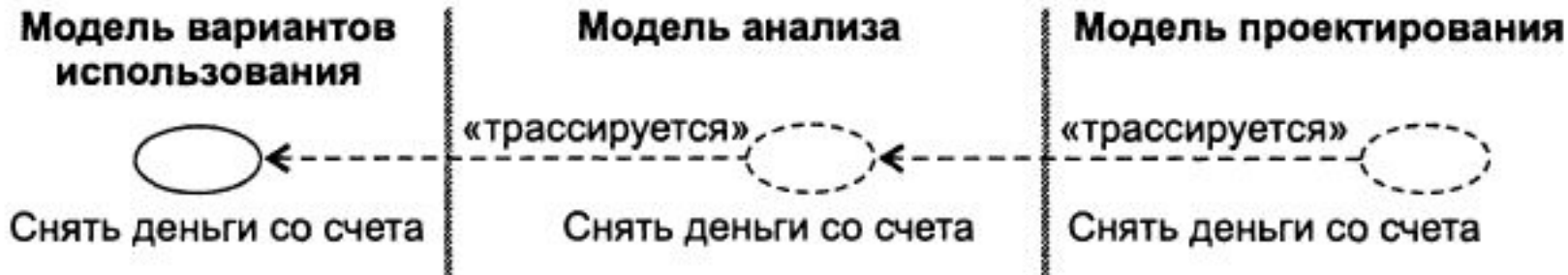


Проектирование ИС

05 2016

Реализации варианта использования в разных моделях



Классы проектирования из модели проектирования трассируются от классов анализа из модели анализа

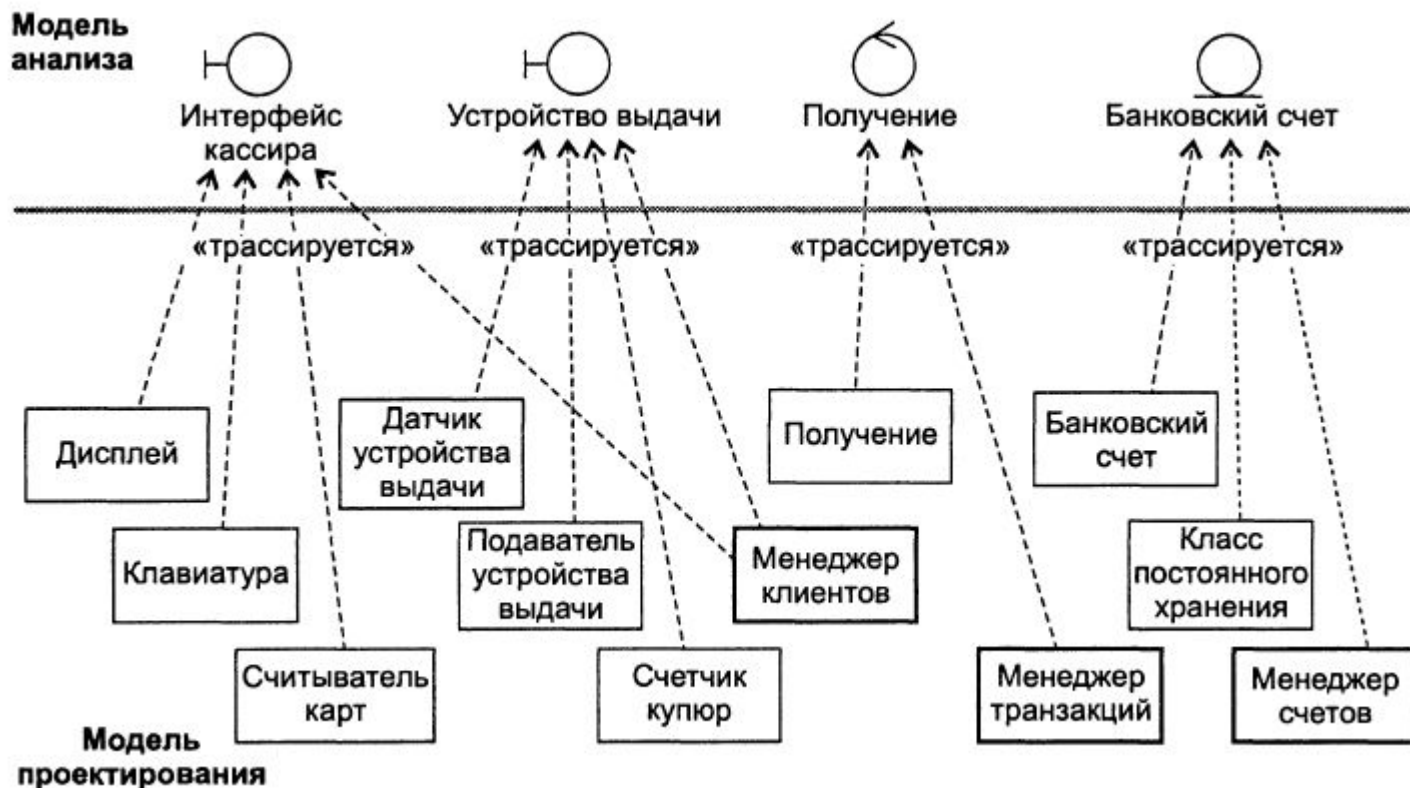
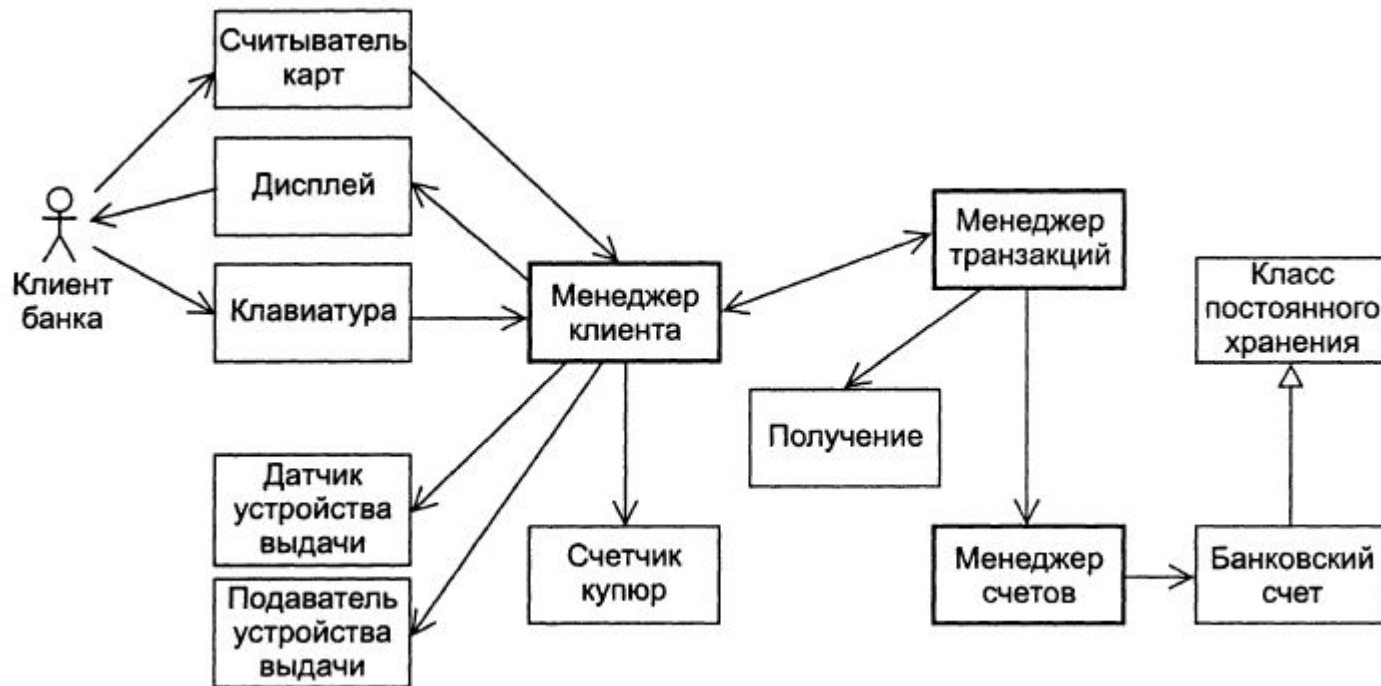


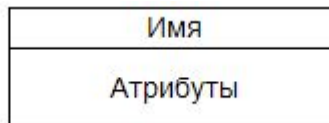
Диаграмма классов проектирования



ДИАГРАММЫ КЛАССОВ

Способы отображения класса

- * **Класс** отображается в виде прямоугольника, который может быть разделен горизонтальными линиями на секции. В этих секциях указывается имя, атрибуты (свойства) и операции (методы)



Пример класса без атрибутов

- * С точки зрения структурного подхода, атрибуты – это переменные, а методы – это функции, описанные в теле класса. Они могут быть доступны или не доступны для изменения (атрибуты) или выполнения (методы) внешними объектами

«interface» <i>iVdopRow</i>
getKmN() getKmK() getVkonLoc() getVkonVag()

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

Видимость

- * Характеризует возможность чтения и модификации значения атрибута объекта описываемого класса, из объектов других классов
- * - "+" – 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.

Символ "/"

- * Символ "/" перед именем атрибута указывает на то, что он является **производным** (т.е. его значение вычисляется из значений других атрибутов или ассоциаций).

Имя

- * **Имя** (name) атрибута представляет собой строку текста, которая используется для его идентификации. Оно должно быть **уникальным** в пределах класса.

Tun

- * **Tun** (type) атрибута выбирается исходя из семантики значений, которые должны храниться в атрибуте, и, как правило, возможностей целевого языка программирования по представлению этих значений. Он 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. Во втором 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. Такой вариант задания кратности характерен для множеств, списков и других атрибутов, допускающих добавление или удаление элементов;
- * - **[0..<число>]** – количество хранимых значений, может быть **не более** указанного числа. Данный вариант применяется при описании массивов фиксированного размера. При этом не обязательно, чтобы все элементы массива имели конкретные значения;
- * - **[0..<число>] [0..<число>]** – применяется при описании двумерных **массивов**. Аналогичным образом можно описать трехмерные, четырехмерные и т.д. массивы.

Исходное значение

- * **Исходное значение** (default value) служит для задания некоторого начального значения атрибута в момент создания отдельного экземпляра класса (объекта).

Модификатор

- * **Модификатор** (modifier) описывает особенности реализации атрибута, например:
- * - **{final}** / **{readOnly}** – 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;
- * - **{unique}** или **{nonunique}** – значения неатомарного атрибута должны быть уникальны или допускаются повторы значений;
- * - **{ordered}** или **{unordered}** – значения неатомарного атрибута должны быть отсортированы или могут содержаться в произвольном порядке;
- * - **{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.

Примеры указания атрибутов

Спецификация атрибута в UML	Генерируемый код для языка Java
+name : String	public String name;
+ pi : double = 3.1415 {final, static}	public final static double pi = 3.1415;
- coordinateXY : int[][] = {{1, 1}, {2, 4}, {3, 9}}	private int[][] coordinateXY = {{1, 1}, {2, 4}, {3, 9}};
# visible : boolean = true	protected boolean visible = true;
- connect : ConnectDB = null;	private ConnectDB connect = null;

3-я секция

- * В третьей секции указывается перечень **методов** класса. Можно выделить шесть основных **типов методов**

Шесть основных *типов* 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 setName(newName : 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 – getName() или определить видимость на экране элемента графического интерфейса – isVisible();
- * - **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.

Описание методов

* [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

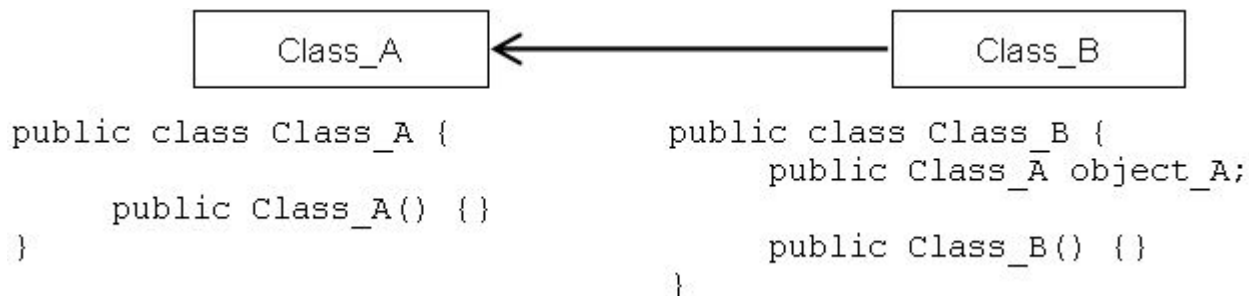
- * **Свойства** служат для указания специфических свойств метода, например:
- * - {native} – реализация метода зависит от платформы (операционной системы);
- * - {abstract} – метод в описываемом классе не имеет тела. Код метода должен быть определен в дочерних классах;
- * - {sequential} – метод допускает только последовательный вызов. Параллельный 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;
- * - {unique} или {nonunique} – возвращает неатомарный атрибут без или с повторами значений;
- * - {ordered} или {unordered} – возвращает отсортированную или неотсортированную последовательность значений неатомарного атрибута;
- * - {seq} / {sequence} – возвращает упорядоченную последовательность значений неатомарного атрибута.

Примеры указания методов

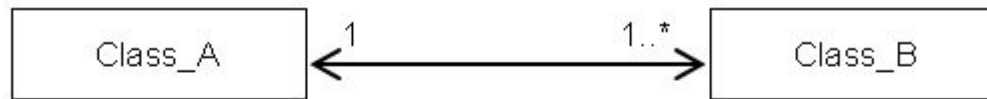
Спецификация метода в UML	Генерируемый код для языка Java
"constructor" + TextFieldInt(value : int, length : int, allignment : int, fontField : Font)	<pre>public TextFieldInt(int value, int length, int allignment, Font fontField) { }</pre>
+ saveData()	<pre>public void saveData() {return;}</pre>
+ isVisible() : boolean	<pre>public boolean isVisible() {return false;}</pre>
# init(text : String, icon : Icon)	<pre>protected void init(String text, Icon icon) {return;}</pre>

Отношение ассоциации

- * **Отношение ассоциации** означает наличие атрибута, в котором будет храниться ссылка (ссылки) на объект (объекты) класса, в сторону которого направлена стрелка ассоциации.



- * Графический символ класса Class_A преобразуется в строки определения самого класса "public class Class_A" и его конструктора "public Class_A() {}". Аналогично для Class_B. Ассоциация от Class_B в сторону Class_A преобразуется в строку "public Class_A object_A;", описывающую атрибут object_A, в котором будет храниться ссылка на объект класса Class_A. Ввиду отсутствия указания кратности отношения, она по умолчанию принимается равной 1.



```
public class Class_A {
    public Class_B object_B[];

    public Class_A() {}
}
```

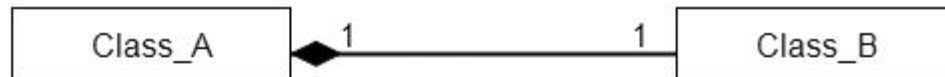
```
public class Class_B {
    public Class_A object_A;

    public Class_B() {}
}
```

- * Наличие двунаправленной ассоциации или ассоциации без стрелок свидетельствует о наличии в обоих классах атрибутов, содержащих ссылки на объекты. Кратность более 1 подразумевает хранение не одной, а нескольких ссылок. Таким образом, один объект класса Class_A будет связан с несколькими объектами класса Class_B. Ссылки на эти объекты будут храниться в массиве object_B[]. Современные CASE-средства позволяют вместо массива указывать другие варианты хранения набора объектов, такие как множества, списки, хешированные таблицы и т.д.

Отношения агрегации и композиции

а

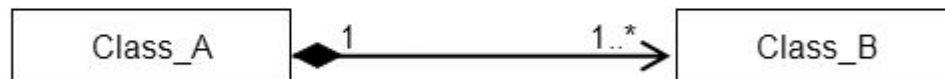


```
public class Class_A {
    public Class_B object_B;

    public Class_A() {}
}
```

```
public class Class_B {
    public Class_B() {}
}
```

б



```
public class Class_A {
    public Class_B object_B[];

    public Class_A() {}
}
```

```
public class Class_B {
    public Class_B() {}
}
```

Отношение обобщения

- * **Отношение обобщения** в тексте программы на языке Java показывается ключевым словом "extends" (расширяет) в дочернем классе.



```
public class Class_A {  
    public Class_A() {}  
}  
public class Class_B extends Class_A {  
    public Class_B() {}  
}
```

Отношение зависимости

- * **Отношение зависимости** не приводит к автоматической генерации кода программы, но свидетельствует об обращении из объекта зависимого класса к атрибутам, методам или непосредственно к объектам независимого класса. Данное отношение в Case-средстве может автоматически отображаться на диаграмме при обратном проектировании или при синхронизации диаграммы и текста программы.

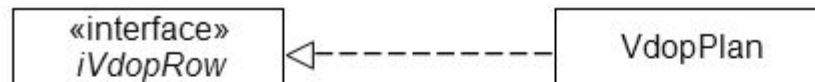


```
public class Class_B {
    ...
    public obrabotka(Class_A object_A) {
        ...
        String name = object_A.name;
        ...
        int age = object_A.getAge();
        ...
    }
}
```

В строке "public obrabotka(Class_A object_A)" используется ссылка на объект класса Class_A. В строке "String name = object_A.name;" выполняется обращение к атрибуту объекта класса Class_A. В строке "int age = object_A.getAge();" выполняется обращение к методу объекта класса Class_A.

Отношение реализации

- * **Отношение реализации** - дополнительное отношение на диаграмме классов по сравнению с диаграммой классов анализа, которое отображается только между классами и интерфейсами. В тексте на языке Java данное отношение обозначается ключевым словом "implements".



```
public interface IVdopRow {  
  
    // спецификации методов  
}
```

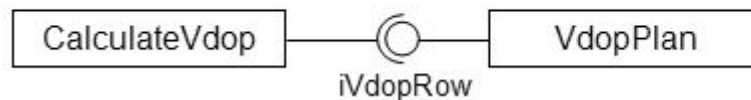
```
public class VdopPlan implements IVdopRow {  
  
    // методы с реализацией  
}
```

Внешний вид отношения подчеркивает тот факт, что оно сочетает в себе особенности обобщения (наследования) и зависимости.

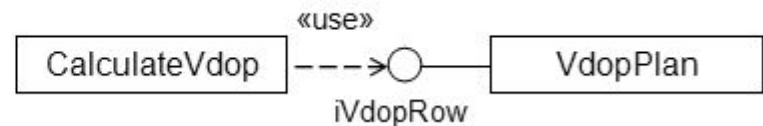
Интерфейс

- * Для отображения интерфейса в UML имеется также другой способ отображения - в виде кружка, который связывается ассоциацией с реализующим его классом. Класс, который использует (англ. use) интерфейс, связывается с ним или ассоциацией с полукругом на конце или зависимостью с соответствующим стереотипом.

a



b



Объекты

: ConnectDB

tipUser = «Администратор»
isConnect = true
nameDB = «Iskra»

Диаграмма классов

- * Является центральным звеном объектно-ориентированного подхода
- * Содержит информацию об объектах системы и статических связях между объектами
- * Отражает *декларативные знания* о предметной области
- * Оперирует понятиями *класса, объекта, отношения, пакета*

Класс

- * **Класс** – это множество объектов, которые обладают одинаковой структурой, поведением и отношениями с объектами из других классов.

Имя_класса

← Простейший вид класса состоит только из секции имени

Имя_класса
атрибуты класса

← Класс с указанием атрибутов (переменных)

Полное описание класса, состоящее из 3 разделов (секций) – секции имени, секции атрибутов, секции операций

Имя_класса
атрибуты класса
операции класса()

Класс

- * *Имя класса* должно быть уникально
- * Имя класса должно начинаться с заглавной буквы.
- * Класс может не иметь экземпляров или объектов. В этом случае он называется **абстрактным классом**, а для обозначения его имени используется *курсив*

Атрибуты класса

- * Атрибут = свойство, которое является общим для всех объектов данного класса
- * Общий формат записи атрибутов:
<квантор видимости> <имя атрибута> [кратность]: <тип атрибута> = <исходное значение> {строка-свойство}

Атрибуты класса.

Квантор видимости

- * Квантор видимости может принимать одно из следующих значений: +, #, -, ~.
- * «+» - атрибут с областью видимости типа **общедоступный** (public).
- * «#» - атрибут с областью видимости типа **защищенный** (protected).
- * «-» - атрибут с областью видимости типа **закрытый** (private).
- * «~» - атрибут с областью видимости типа **пакетный** (package).

Атрибуты класса.

Имя атрибута

- * Представлено в виде *уникальной* строки текста
- * Имя атрибута является единственным обязательным элементом в синтаксическом обозначении атрибута
- * Должно начинаться со строчной буквы
- * По практическим соображениям записывается без пробелов

Атрибуты класса.

Кратность атрибута

- * *Кратность атрибута* характеризует общее количество конкретных атрибутов данного типа, входящих в состав отдельного класса.
- * Формат: [нижняя граница .. верхняя граница]
- * Примеры: [0..1], [0..*], [1..3,5..7]

Атрибуты класса. Тип атрибута

- * Выражение, определяемое некоторым типом данных (например, в зависимости от языка программирования)
- * В простейшем случае – осмысленная строка текста.
- * Пример:

цвет: Color

имяСотрудника[1..2]: String;

видимость: Boolean

Атрибуты класса. Исходное значение

* Служит для задания некоторого начального значения в момент *создания* отдельного экземпляра класса

* Пример:

цвет: *Color* = (255, 0, 0)

имяСотрудника[1..2]: *String* = 'Иван Иванов';

видимость: *Boolean* = истина

Атрибуты класса. Строка-свойство

- * Служит для указания **дополнительных свойств атрибута**, которые могут характеризовать особенности изменения значений атрибута в ходе выполнения соответствующей программы.
- * Это значение принимается за **исходное значение атрибута**, которое не может быть изменено в дальнейшем.
- * Пример:
заработнаяПлата: Currency = \$500 {frozen}

Операции класса

- * Представляют собой некоторый сервис, который предоставляет каждый экземпляр класса или объект по требованию своих клиентов.
- * Правила записи операций:
<квантор видимости> <имя операции> (список параметров): <выражение типа возвращаемого значения> {строка-свойство}

Операции класса. Список параметров

* Список параметров является перечнем разделенных запятой формальных параметров, каждый из которых, в свою очередь, может быть представлен в следующем виде:

*<вид параметра> <имя параметра> : <выражение типа> =
<значение параметра по умолчанию>*

Операции класса. Строка-свойство

- * **Строка-свойство** служит для указания значений свойств, которые могут быть применены к данной операции.
- * Например, для указания последовательности действий будет использована строка-свойство вида:

{concurrency = имя} ,

где *имя* может принимать одно из следующих значений:

- * **sequential** (последовательная),
- * **concurrent** (параллельная),
- * **guarded** (охраняемая)

Операции класса. Примеры

- * +нарисовать (форма : Многоугольник = прямоугольник, цветЗаливки : Color = (0, 0, 255));
- * -изменитьСчетКлиента (номерСчета : Integer) : Currency;
- * #выдатьСообщение() : ('Ошибка деления на ноль').

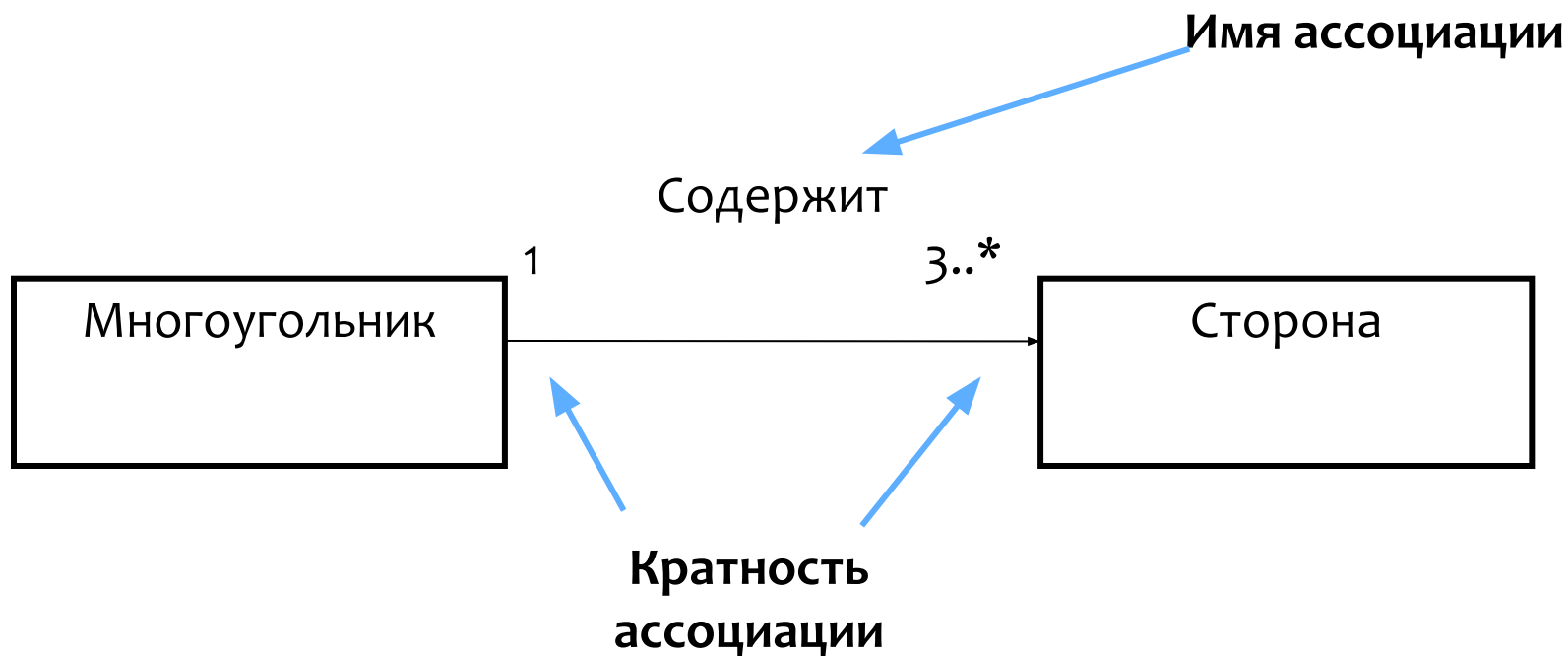
Отношения между классами

Базовыми отношениями на диаграмме классов являются:

- * отношения **ассоциации** (*association*);
- * отношения **обобщения** (*generalization*);
- * отношения **агрегации** (*aggregation*);
- * отношения **композиции** (*composition*);
- * отношения **зависимости** (*dependency*).

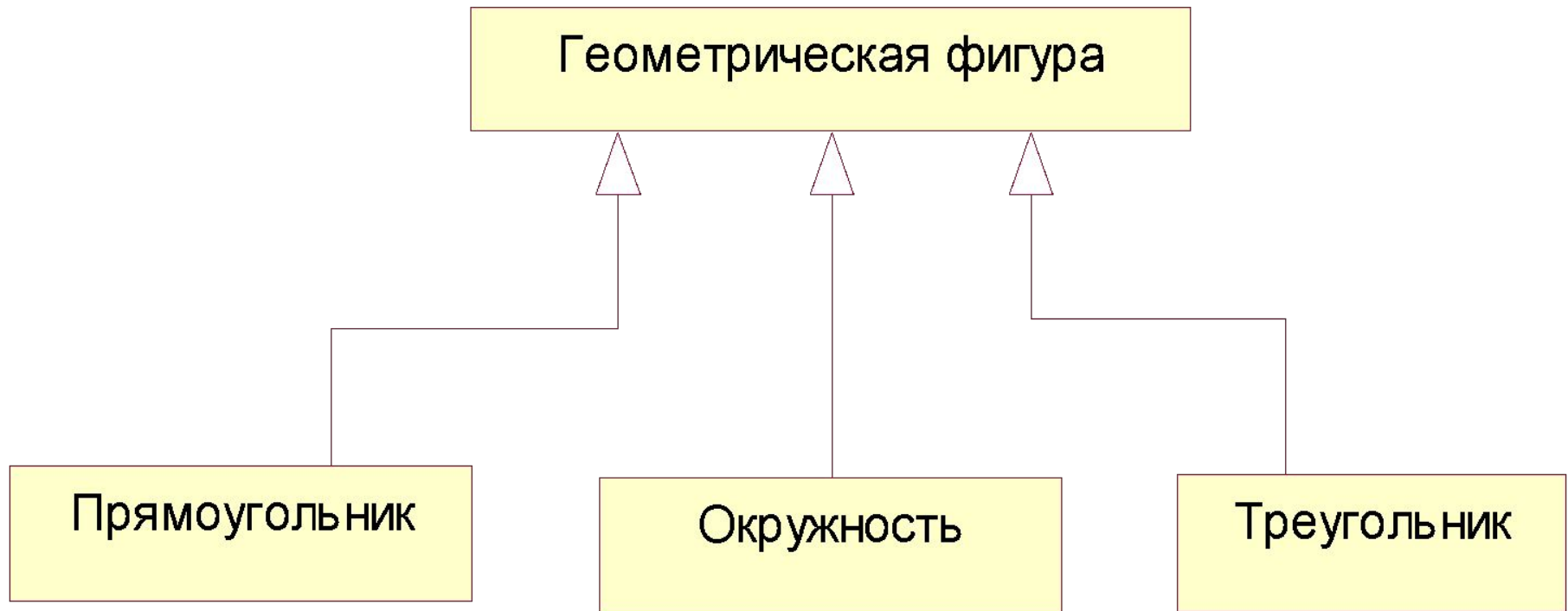
Отношение ассоциации

- * **Отношение ассоциации** свидетельствует о наличии произвольного отношения между классами.



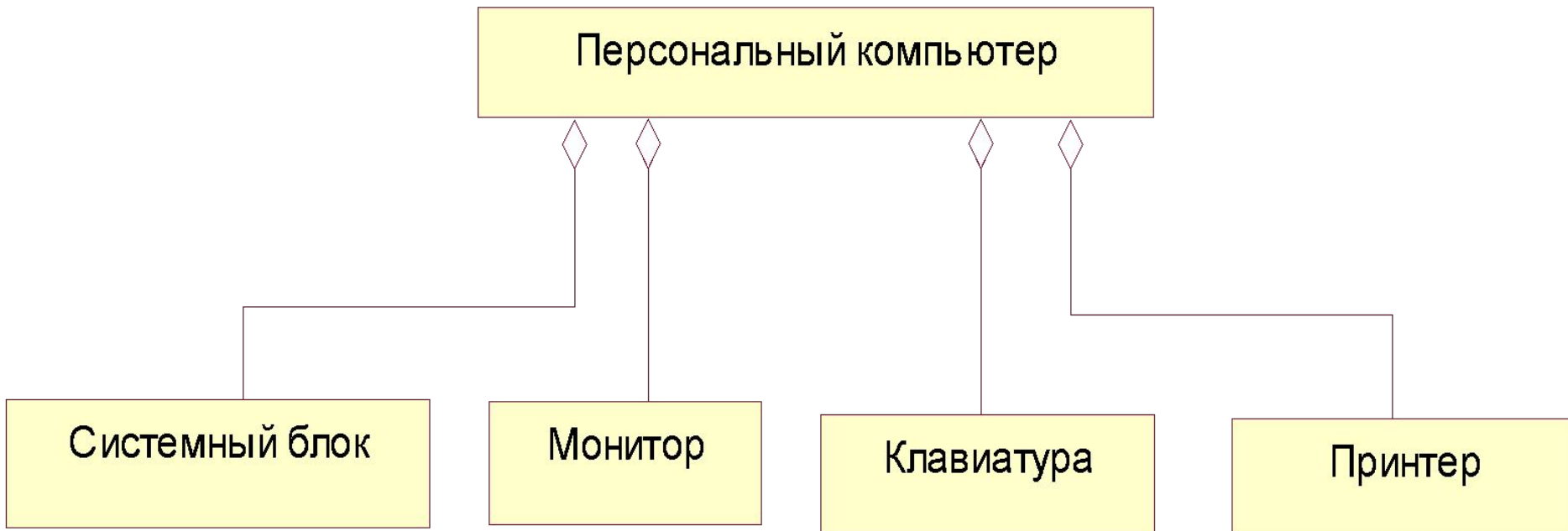
Отношение обобщения

Является отношением классификации между более общим элементом (родителем или предком) и более частным или специальным элементом (дочерним или потомком)



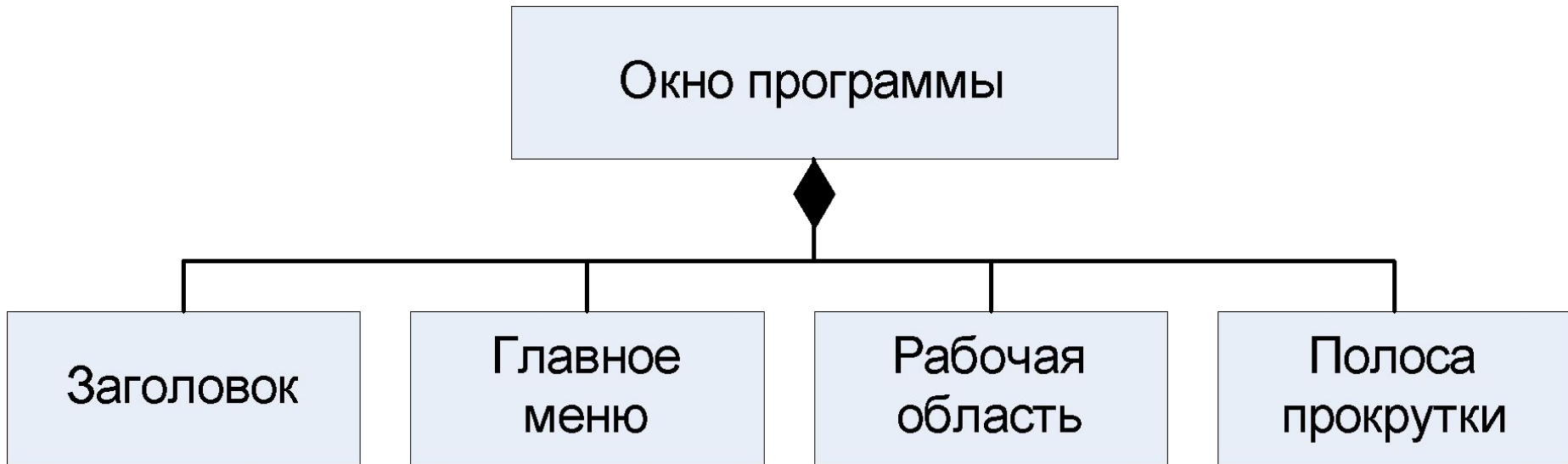
Отношение агрегации

- * Смысл: один из классов представляет собой некоторую сущность, которая **включает** в себя в качестве составных частей другие сущности.
- * Применяется для представления системных взаимосвязей типа «часть-целое».



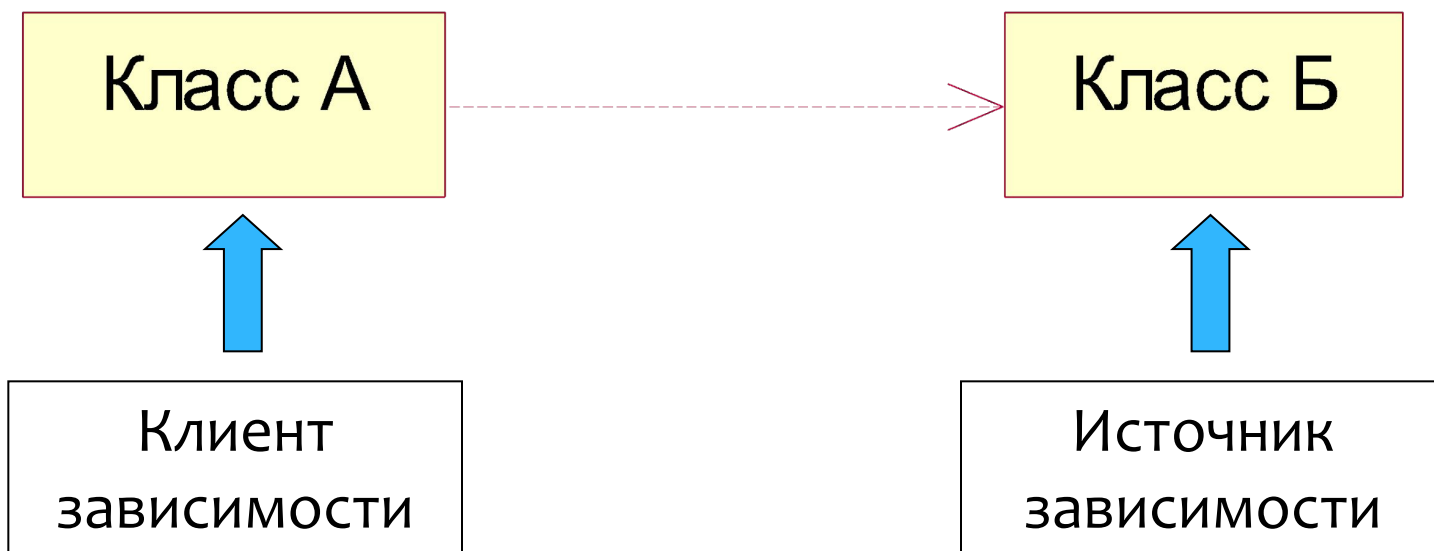
Отношение композиции

- * Является частным случаем отношения агрегации.
- * Части не могут выступать в отрыве от целого, т.е. с уничтожением целого уничтожаются составные части.



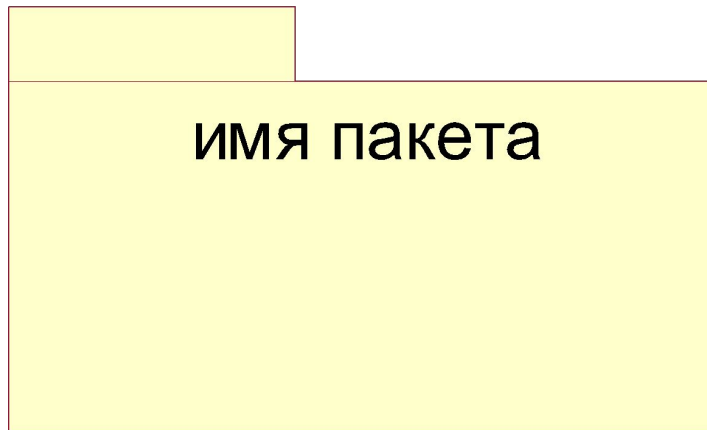
Отношение зависимости

- * Используется в такой ситуации, когда некоторое изменение одного элемента модели может потребовать изменения другого элемента.

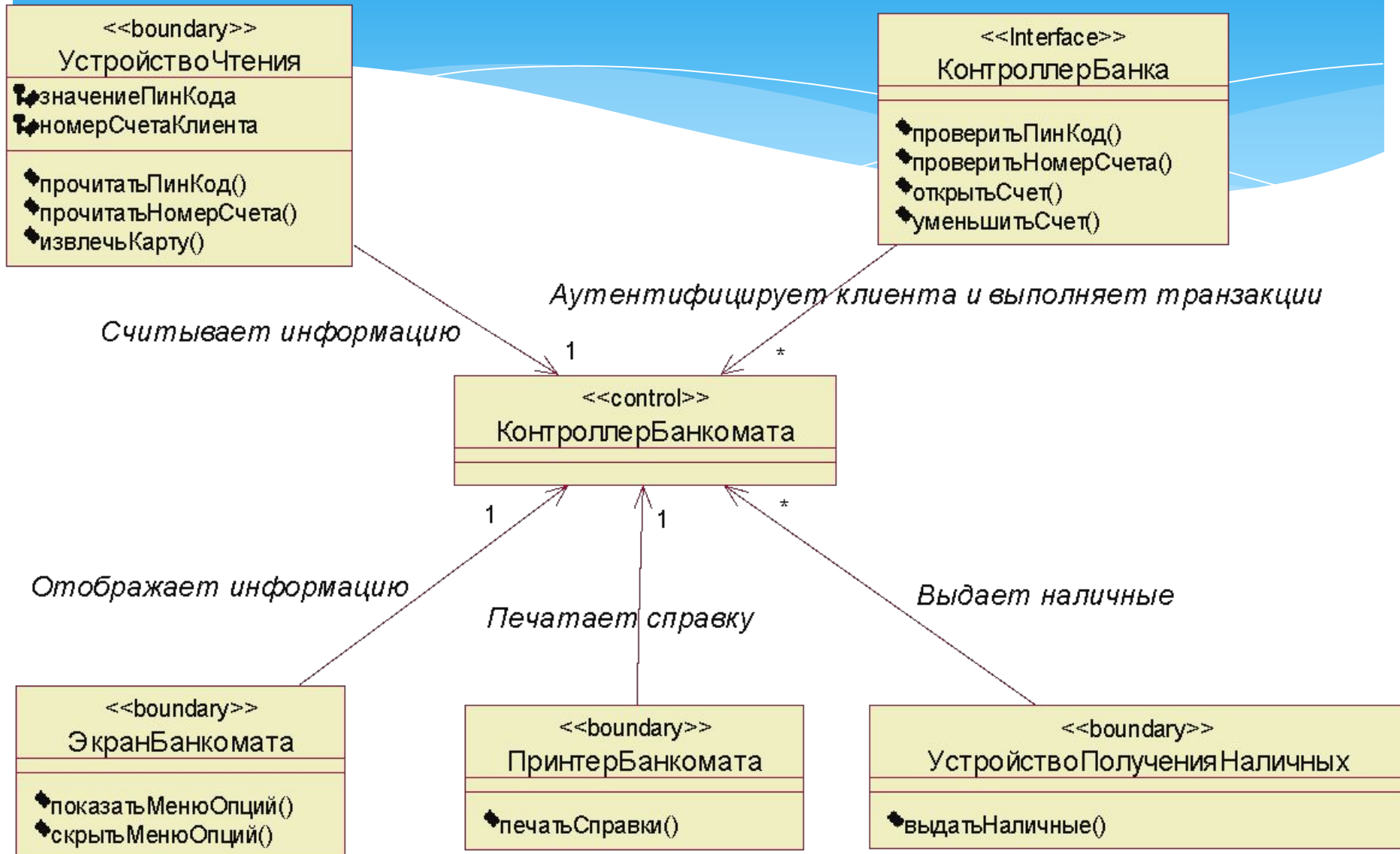


Пакеты

- * служат для **группировки** элементов модели
- * Любой пакет владеет своими элементами
- * любой элемент может принадлежать *только одному пакету*



Пример диаграммы классов



Расширения языка UML

Расширения языка
UML

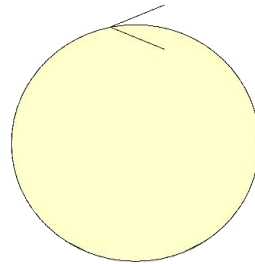
```
graph TD; A[Расширения языка UML] --> B[Профиль для процесса разработки ПО (The UML Profile for Software Development)]; A --> C[Профиль для бизнес-моделирования (The UML Profile for Business Modeling)];
```

Профиль для процесса
разработки ПО
(The UML Profile for
Software Development)

Профиль для бизнес-
моделирования (The UML
Profile for Business
Modeling)

Профиль для процесса разработки ПО

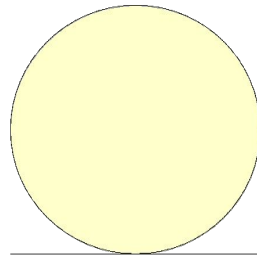
- * **Управляющий класс (control)** – отвечает за координацию действий других классов.



NewClass

Профиль для процесса разработки ПО

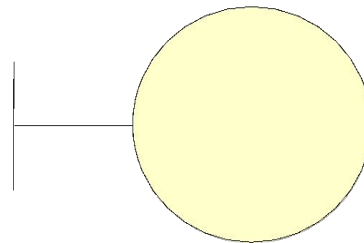
- * **Класс-сущность (entity)** содержит информацию, которая должна храниться **постоянно** и не уничтожаться с уничтожением объектов данного класса или прекращением работы моделируемой системы.



NewClass2

Профиль для процесса разработки ПО

- * **Граничный класс (boundary)** – располагается на границе системы с внешней средой, но является составной частью системы.



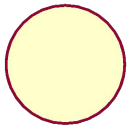
NewClass3

Задание

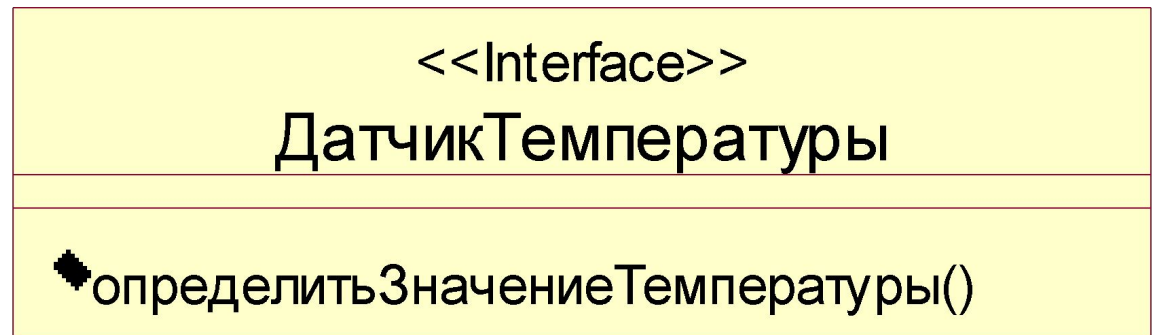
- * изучить самостоятельно графические примитивы профиля бизнес-моделирования.

Интерфейс (interface)

- * в контексте языка UML является специальным случаем класса, у которого имеются только операции и отсутствуют атрибуты.



ДатчикТемпературы



Рекомендации

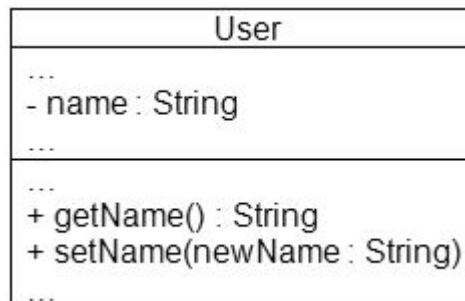
- * 1. За основу диаграммы классов при ее разработке берется диаграмма классов анализа.
- * 2. Для классов должны быть определены и специфицированы все атрибуты и методы. Их спецификация, как правило, выполняется с учетом выбранного языка программирования.
- * 3. При определении методов рекомендуется использовать сообщения с ранее разработанных диаграмм последовательности и коммуникации.

- * 4. Детальное проектирование граничных классов, как правило, не требуется. Большинство современных средств разработки поддерживает визуальную разработку интерфейса системы – меню, диалоговых форм, элементов диалоговых окон, панелей инструментов и т. д. В качестве исходных данных для их проектирования служат прототипы пользовательских интерфейсов. В связи с этим 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. Современные CASE-средства позволяют разрабатывать такие диаграммы и синхронизировать их с 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.

- * 9. Для атрибутов рекомендуется назначать видимость `private` (закрытый) или `protected` (защищенный). Если требуется чтение значения такого атрибута из объектов других классов, то следует предусмотреть для него `get`-метод, а если возможность установки нового значения – `set`-метод.



- * 10. Для методов видимость `public` (общедоступный) следует устанавливать только в случае крайней необходимости.
- * 11. Ввиду большого количества классов в системе рекомендуется диаграммы классов разрабатывать отдельно для каждого пакета. По умолчанию Case-средства поддерживают именно такой подход проектирования, хотя и допускают разработку диаграмм, на которых присутствуют классы из разных пакетов.
- * 12. При проектировании диаграммы и отдельных классов рекомендуется пользоваться шаблонами проектирования.

