Объектно-ориентированные CASE-технологии
Язык UML 2.5.

Владимир Юрьевич Романов,
Московский Государственный Университет им. М.В.Ломоносова
Факультет Вычислительной Математики и Кибернетики
vromanov@cs.msu.su,
vladimir.romanov@gmail.com
http://master.cmc.msu.ru

UML – Unified Modeling Language. Унифицированный язык моделирования

- Стандарт на язык моделирования разработанный консорциумом фирм Object Management Group:

  http://www.omg.org

- Стандартизация языка UML консорциумом OMG:
  http://www.omg.org/uml
  http://www.uml.org/

- Текущие версии стандарта доступные для свободного скачивания:
  http://www.omg.org/spec/

UML – Unified Modeling Language. Стандарты связанные с языком UML

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2024

MODELING AND METADATA SPECIFICATIONS

UML, MOF, CWM, XMI Specifications

SPECIFICATION	acronym	topical area / domain	Document #
Action Language for Foundational UML	ALF	modeling	ptc/2012-08-43
Common Terminology Services 2	cts2	modeling	formal/2012-11-01
Common Warehouse Metamodel	CWM	data warehousing, modeling	formal/2003-03-02
CWM Metadata Interchange Patterns	MIPS	data warehousing, modeling	formal/2004-03-25
Diagram Definition	DD	modeling	formal/12-07-01
Essence - Kernel and Language for Software Engineering Methods	Essence	modeling	ptc/2013-06-08
Interaction Flow Modeling Language	IFML	modeling	ptc/2013-03-08
Meta Object Facility Core	MOF	modeling	formal/2011-08-07
Model Driven Message Interoperability	MDMI	modeling	formal/2010-03-01
Model-level Testing and Debugging	MLTD	modeling	ptc/2007-05-14
MOF Model to Text Transformation Language	MOFM2T	modeling	formal/2008-01-16
MOF Query / View / Transformation	QVT	modeling	formal/2011-01-01
MOF Support for Semantic Structures	SMOF	modeling	formal/2013-04-02
MOF 2 Facility and Object Lifecycle	MOFFOL	modeling	formal/2010-03-04
MOF 2 Versioning and Development Lifecycle	MOFVD	modeling	formal/2007-05-01
Object Constraint Language	OCL	modeling	formal/2012-01-01
OMG Systems Modeling Language	SysML	modeling	formal/2012-06-01
Ontology Definition Metamodel	ODM	modeling	formal/2009-05-01
Reusable Asset Specification	RAS	modeling	formal/2005-11-02
Semantics of a Foundational Subset for Executable UML Models	FUML	modeling	formal/2011-02-01
Service oriented architecture Modeling Language	SoaML	modeling	formal/2012-05-01
Software Process Engineering Metamodel	SPEM	modeling	formal/2008-04-01
Unified Modeling Language	UML	modeling	formal/2011-08-05, formal/2011-08-06
UML Simplified (UML 2.5)	UML	modeling	ptc/2012-10-24
UML Human-Usable Textual Notation	HUTN	modeling	formal/2004-08-01
MOF 2 XMI Mapping	XMI	modeling	formal/2011-08-09

UML – Unified Modeling Language.
Инфраструктура и суперструктура языка UML

- Текущая версия языка **UML**:
  http://www.omg.org/spec/UML/2.5/PDF/

- **O**bject **C**onstraint **L**anguage™ (OCL™)
  http://www.omg.org/spec/OCL/2.4/PDF/

- **D**iagram **D**efinition (**DD**)
  http://www.omg.org/spec/DD/1.1

- **X**ML **M**etadata **I**nterchange (**XMI**) формат для обмена моделями инструментами:
  http://www.omg.org/spec/XMI/2.5.1/

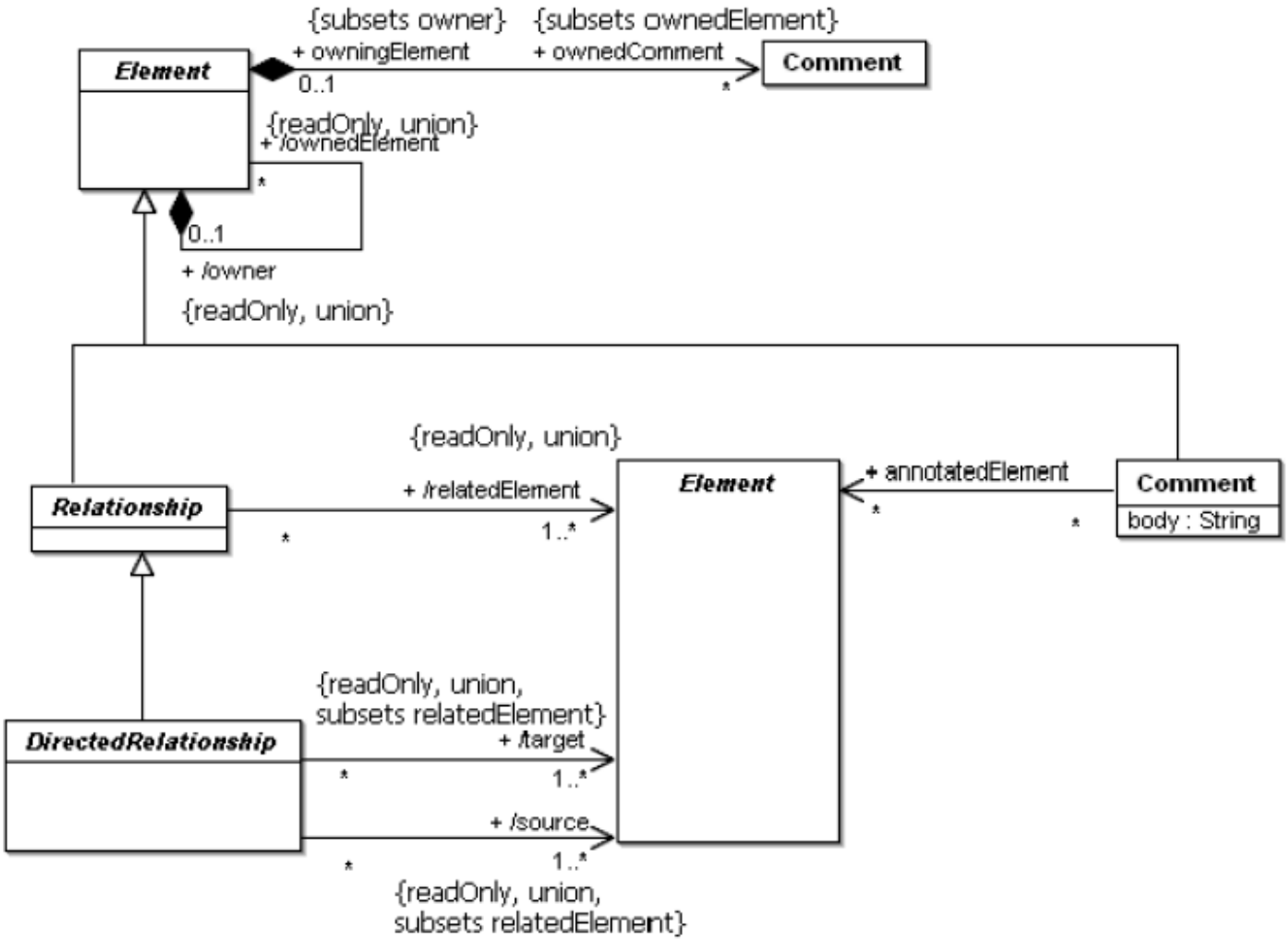
- Спецификация **UML** в формате **XMI**
  http://www.omg.org/spec/UML/20161101/UML.xmi

UML – Unified Modeling Language. Объектный язык ограничений (OCL)

- **O**bject **C**onstraint **L**anguage (**OCL**)
  https://www.omg.org/spec/OCL/2.4/

- Спецификация языка OCL на языке XML
  http://www.omg.org/spec/OCL/20090501/OCL.cmof

UML – Unified Modeling Language.
Фрагмент спецификации метамодели UML



UML – Unified Modeling Language.
Фрагмент спецификации метамодели OCL

- **if** memberEnd->size() > 2 **then** ownedEnd->*includesAll*(memberEnd)

- parents()->*select*(oclIsKindOf(Association)).oclAsType(Association)->
  forAll(p | p.memberEnd->size() = **self**.memberEnd->size())

- *Sequence*{1..**self**.memberEnd->size()}->
  forAll(i |
  self.general->*select*(oclIsKindOf(Association)).oclAsType(Association)
  ->
  forAll(ga |
  self.memberEnd->
  at(i).type.conformsTo(ga.memberEnd->at(i).type)))

UML – Unified Modeling Language.
Meta Object Facility (MOF)

- **M**eta **O**bject **F**acility (**MOF**)
  http://www.omg.org/spec/MOF/2.5.1/

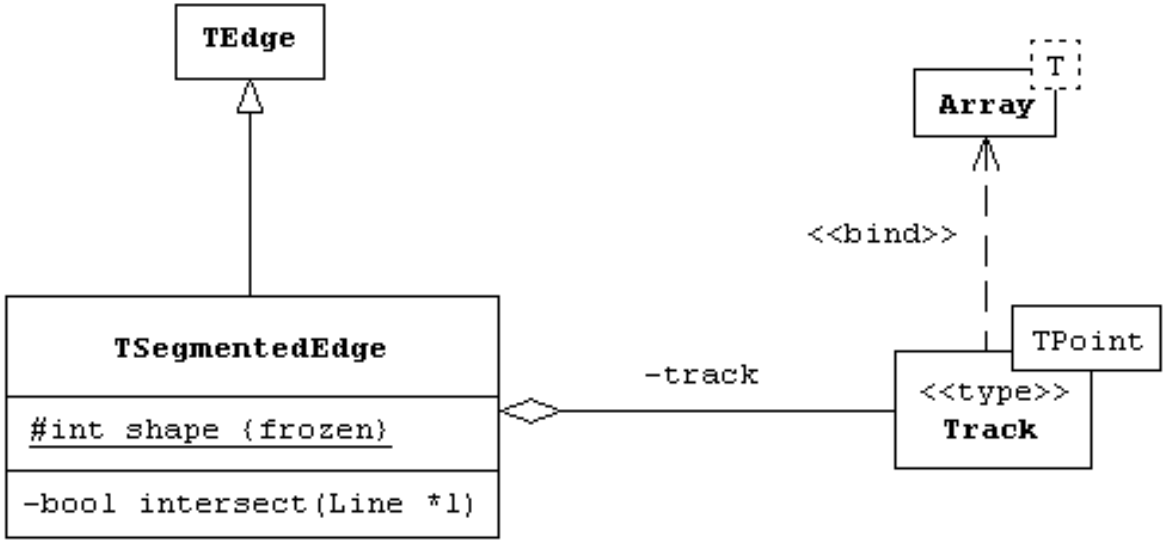
- Спецификация MOF на языке XMI
  http://www.omg.org/spec/MOF/20131001/MOF.xmi

UML – Unified Modeling Language. Практические занятия по курсу

- **Решение задач**
  - Преобразование UML диаграмм в текст программы на языке программирования (Java | C# | C++)
  - Преобразование текста программы на языке программирования (Java | C# | C++) в диаграмму UML

UML – Unified Modeling Language. Практические занятия по курсу

- **Задача 1.** Представить следующую диаграмму на языке UML в виде программы на языке C++.



UML – Unified Modeling Language. Практические занятия по курсу


```
class Graph {
    public Nodes nodes;
    public Edges edges;
    private Diagram d;
    public Graph() {
        nodes = new Nodes();
        edges = new Edges();
    }
    public void draw(Diagram d) {
        geometry.Rect R = d.getArea();
        nodes.draw(d);
        edges.draw(d);
        r.draw(d);
    }
    static public void draw(Diagram d) {
        d = new Diagram();
        Graph g = new Graph();
        g.draw(d);
    }
} // class Graph
```

UML – Unified Modeling Language. Совместные проекты по курсу (1)

Совместный проект 1

- Используя механизм рефлексии языка Java разработать программу строящую UML-модель исполняемого файла языка Java, а затем используя эту модель, строящую UML-диаграмму наследования классов в модели. Для визуализации UML-диаграммы использовать векторную графику формата HTML5 и SVG.

UML – Unified Modeling Language. Совместные проекты по курсу (2)

Совместный проект 2

- Используя механизм JDBC языка Java для анализа метаданных базы данных построить UML модель структуры базы данных и визуализировать ее с помощью графической нотации языка UML в форматах HTML5 и SVG

Литература (1)

1. Object Management Group,
   UML 2.5 Superstructure Specification,
   OMG document ptc-06-04-02.pdf
   www.omg.org/uml

2. Буч Г., Якобсон А., Рамбо Дж., Орлов С.А.
   UML. Классика CS. 2-е изд. 2005 год.
   ISBN 5-469-00599-2

3. International Standard ISO/IEC 14482.
   Programming Languages – C++.

4. Бьерн Страуструп. Язык программирования C++.
   Издательство Бином. Москва. 1999.
   ISBN 5-7989-0127-0.

Литература (2)

5. James Gosling, Bill Joy, Guy Steele, Gilad Bracha. The Java™ Language Specification. Third Edition. ISBN 0-321-24678-0

6. Брюс Эккель. Философия Java. 3-е издание. Издательство «Питер». Петербург 2003. ISBN 5-88782-105-1

7. Standard ECMA-334 3rd Edition / June 2006 C# Language Specification. www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf

8. Эндрю Троелсен. C# и платформа .NET. Издательство «Питер». Петербург 2002. ISBN 5-318-00750-3

Роль графической нотации языка UML

- Нотация языка UML – стандартная нотация современных CASE-инструментов.
- Семантика языка UML (метамодель языка UML)– описывается с помощью нотации языка UML.
- Нотация языка UML широко используется в литературе для описания структуры и поведения программного обеспечения
- В данном курсе нотации UML используется для сравнительного изучения понятий языка моделирования UML и языков программирования C#, Java, C++.

Разновидности UML-диаграмм. Обзор назначения диаграмм (1)

- Диаграмма статической структуры
- Диаграмма прецедентов.
- Диаграмма коммуникации объектов.
- Диаграмма последовательности взаимодействия объектов.

Разновидности UML-диаграмм. Обзор назначения диаграмм (2)

- Диаграмма состояний и переходов.
- Диаграмма деятельности.
- Диаграмма модулей и компонентов.
- Диаграмма внедрения.

Диаграммы статической структуры

1. Диаграмма статической структуры. Классификаторы и пакеты

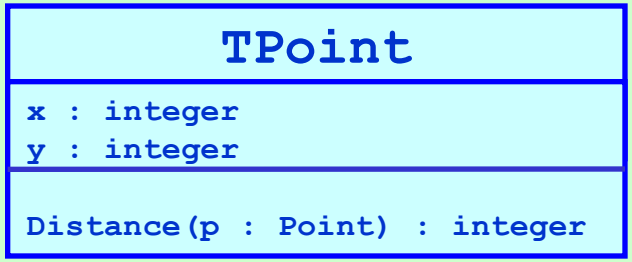
- Классификаторы и пакеты на диаграмме статической структуры
- Отношения между классификаторами и пакетами на диаграмме статической структуры

1.1 Классификаторы

- Классификаторы
- Свойства классификаторов
- Пакеты и их свойства
- Шаблоны для классификаторов и функций

Варианты изображения классов в графической нотации UML

Варианты изображения классов в графической нотации UML



Классы в нотации языков UML, C++, Java и C#


```
case::geometry::TPoint
```

Квалифицированное имя класса

```
// C#
namespace case.geometry {
    public class TPoint {
        int x, y;
    }
}
```

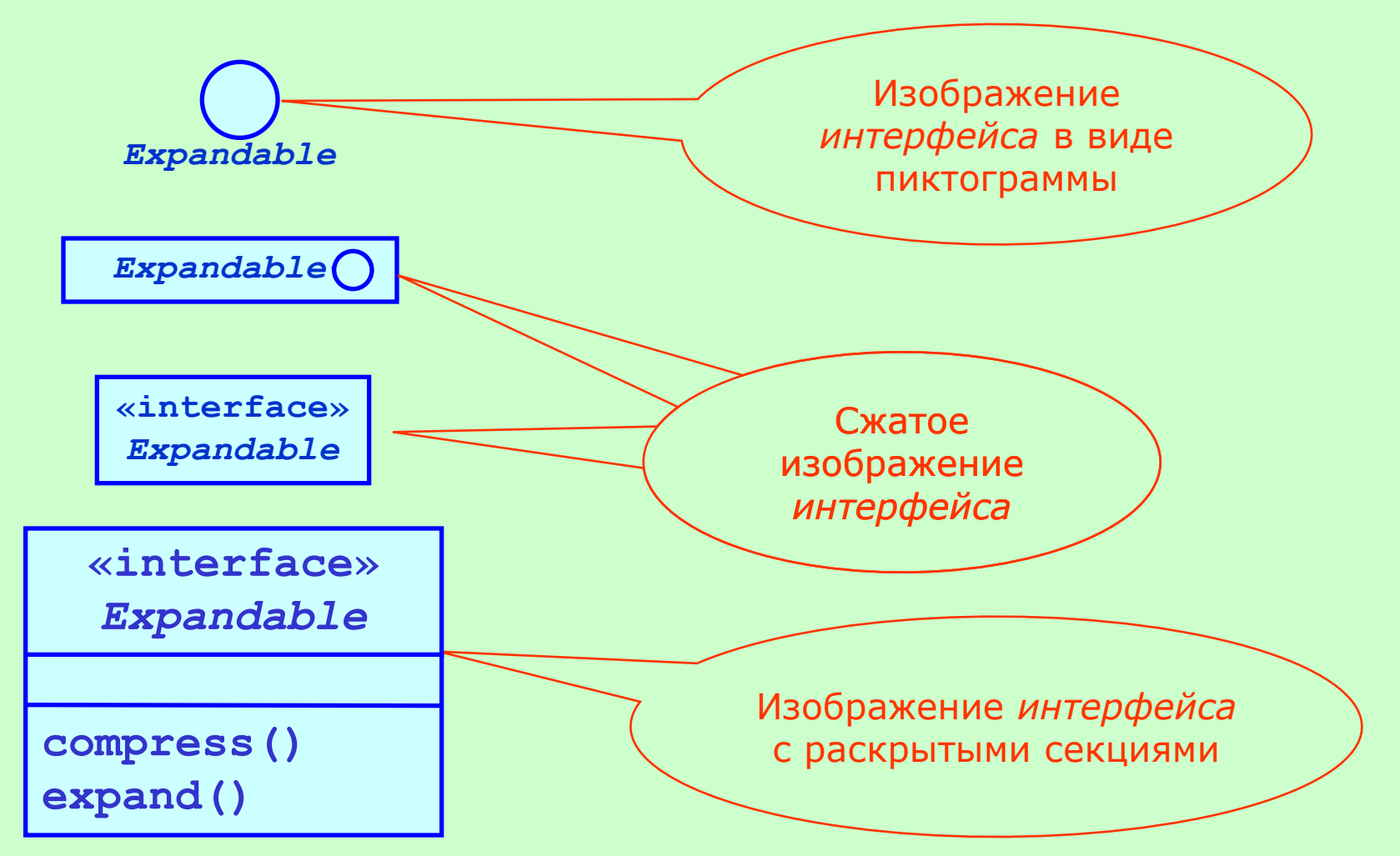
```
// Java
package case.geometry;

public class TPoint {
        int x, y;
}
```

```
// C++
namespace case {
    namespace geometry {
        class TPoint {
            int x, y;
        };
    }
}
```

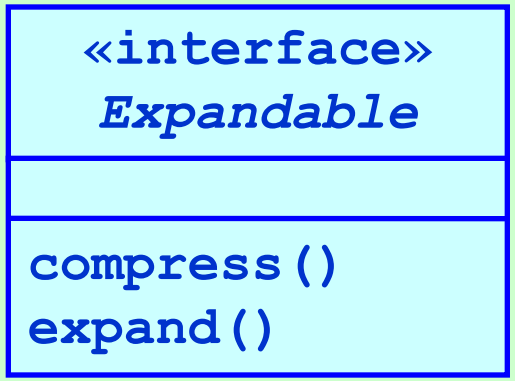
Назначение интерфейсов

- Интерфейс – предоставляет спецификацию множества согласованных операций. Метафора – «*что умеет делать*» класс реализующий данный интерфейс.
- Позволяет отделить спецификацию операций от их реализации. Спецификация одна, реализаций может быть много.
- Позволяет строить независимые иерархии наследования для интерфейсов и классов.
- Позволяет реализовывать более универсальные алгоритмы, работающие с интерфейсами. Алгоритмы применимы к множеству классов реализующих интерфейсы. Эти алгоритмы могут быть унаследованы в классах-потомках.

Варианты изображения интерфейсов в графической нотации UML



Интерфейсы в нотации языков UML, C++, Java и C#



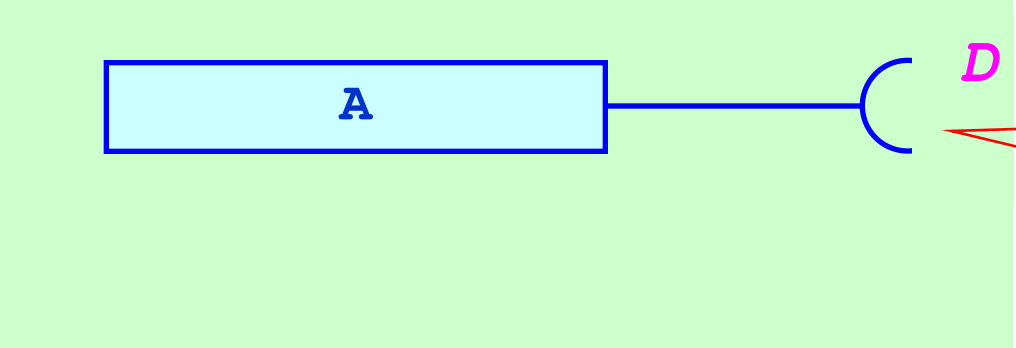

```
// C#
public interface Expandable {
    void compress();
    void expand();
}
```

```
// Java
public interface Expandable {
    public void compress();
    public void expand();
}
```

```
// C++
class Expandable {
public:
    virtual void compress() = 0;
    virtual void expand() = 0;
};
```

Чистые виртуальные функции

Нотация «использование интерфейса» в языке UML 2.1



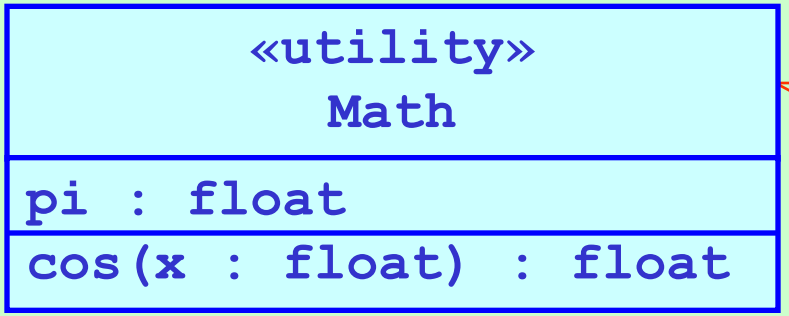
Класс **A** использует интерфейс **D**

```
// Java
import D;

public class A {
    public D getD() {
        // …
    }
public void modify(D d) {
        // …
    }
}
```

Пример на языке Java: Класс **A** использует интерфейс **D**

```
// Java
public interface D {
}
```

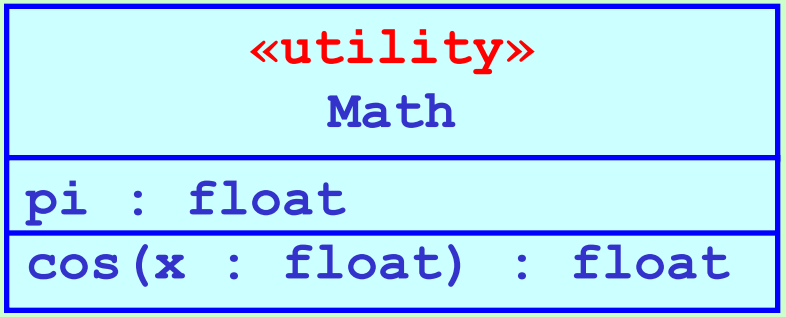
Утилиты в графической нотации UML. Изображение и назначение



Изображение *утилиты* с раскрытыми секциями

- Назначение - группирование и именование части используемых констант и операций общих для множества классификаторов.
- Упрощается доступ к константам и операциям через имя утилиты.
- Не существует экземпляров утилиты.
- Реализуется с помощью статических атрибутов и операций

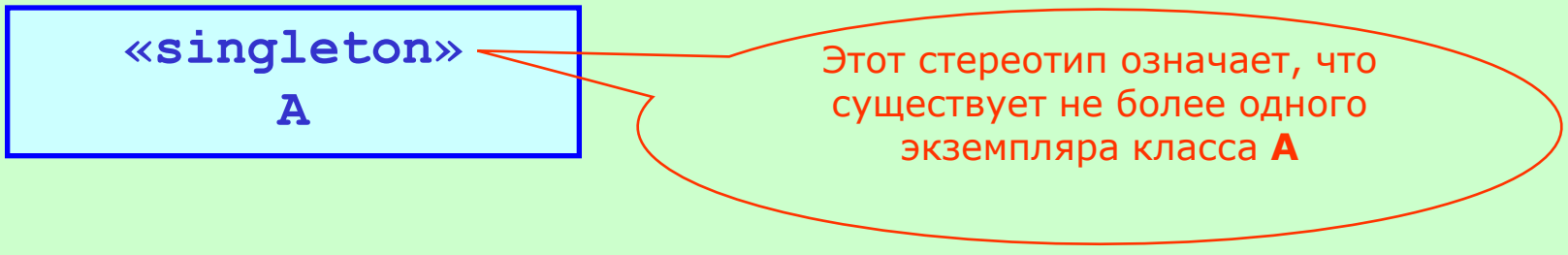
Утилиты в графической нотации UML. Реализация утилит




```
// Java
public class Math {
   final public static float pi = 3.14;
         public static float sin(float x) { … }
}
```

```
// C++
class Math {
public:
   static const float pi = 3.14;
   static float sin(float x) { … };
};
```

Синглетон в графической нотации UML. Изображение, назначение и реализация



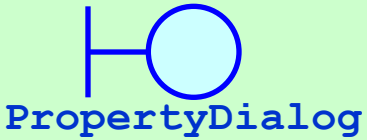

```
// Java
public class A {
   static private A instance;

   private A() {}

   static public A getA() {
       if (instance = null)
           instance = new A();
       return instance;
   }
}
```

Типы в графической нотации UML. Изображение, назначение и реализация

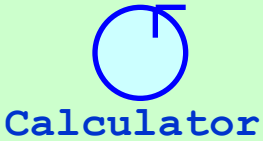

```
// C++
typedef char* String;

typedef std::vector<TPoint> Path;
```

- Назначение описания типов в языке C++ – локализация и упрощение модификации определений типов.

Классификаторы стадии анализа требований к системе



<<boundary>> ***Граничный классификатор***.
Через граничные классификаторы система взаимодействует с внешней средой.



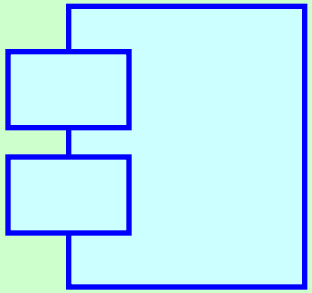
<<control>> ***Управляющий классификатор***.
Управляющие классификаторы реализуют логику работы системы.

<<entity>> ***Классификатор - Сущность.***
Сущности локализуют зависимость системы от способов хранения информации.

Назначение классификаторов стадии анализа:
построение 3-х уровневой модели системы устойчивой к изменениям:

- в способах взаимодействия системы с внешней средой
- в способах хранения информации.

Классификаторы – Компоненты



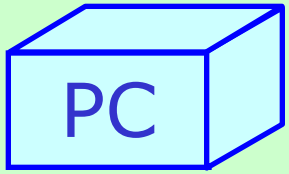
King.java

Компонент – представляет

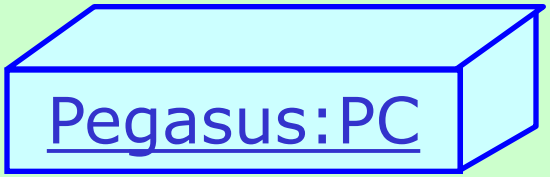
- тексты программ
- исполняемые файлы на интерпретируемых языках
- исполняемые двоичные файлы
- файлы с двоичными данными
- таблицы баз данных

- Назначение компонент – моделирование физической реализации системы. Далее компоненты будут рассмотрены на UML-диаграммах реализации (Implementation Diagrams).

Классификаторы – Вычислительные Узлы



Вычислительный узел – классы вычислительных узлов моделируемой системы

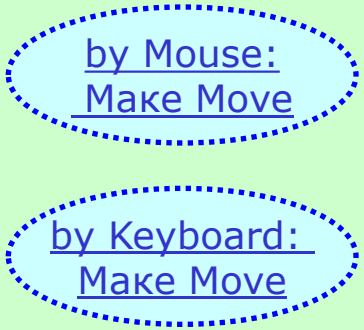


Экземпляр вычислительного узла – элемент для построения вычислительных сетей моделируемой системы

• Назначение *вычислительных узлов* – моделирование физической реализации системы. Далее вычислительные узлы будут рассмотрены на *UML-диаграммах реализации*.

Классификаторы – Прецеденты

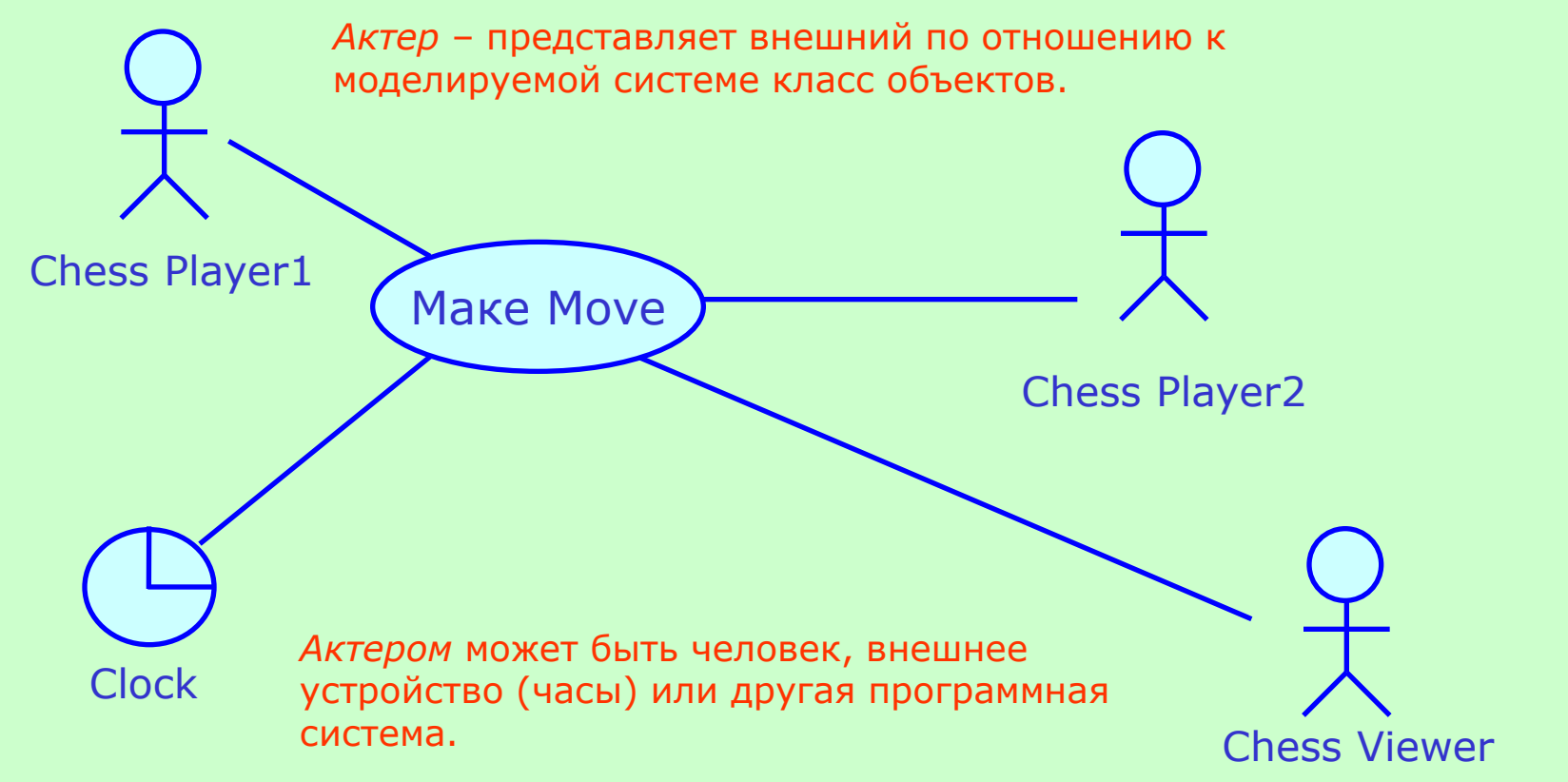
Прецедент – представляет функциональность системы. Например, перемещение элемента диаграммы.



Экземпляр прецедента –
один из способов реализации функциональности моделируемой системы

• Назначение *прецедентов* (использования системы): моделирование взаимодействия системы с внешней средой.
В дальнейшем прецеденты будут рассмотрены на *UML-диаграммах прецедентов (Use Case diagrams).*

Классификаторы – Актеры



• Назначение *актеров* – моделирование взаимодействия системы с внешней средой. Актеры взаимодействуют с *прецедентами* использования системы.

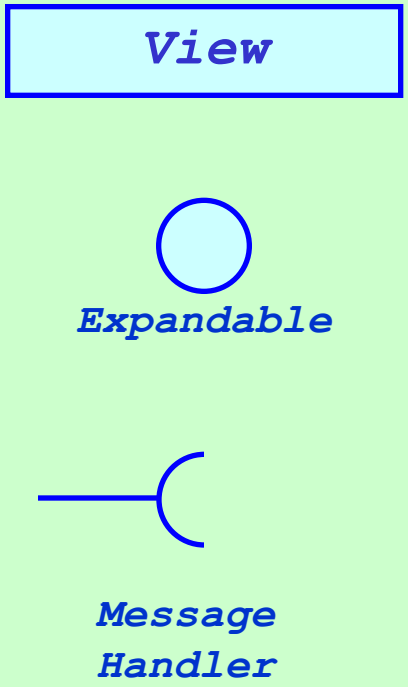
1.2 Свойства классификаторов

- Классификаторы
- Свойства классификаторов
- Пакеты и их свойства
- Шаблоны для классификаторов и утилит

Графические обозначения свойств классификаторов (курсив)

Абстрактность классификатора означает невозможность создания экземпляров данного классификатора.

Абстрактность на диаграмме обозначается курсивом.



Абстрактный *класс*

Интерфейс

Использование интерфейса

Графические обозначения свойств классификаторов (толщина линии)

Активный класс – класс, экземпляры которого выполняются на отдельном потоке управления. В качестве такого потока управления могут использоваться *процесс (process)* или *нить (thread)*.

Активные классы обозначаются толстой рамкой.

В языке Java *активные классы* - это классы-потомки класса `Thread` или классы реализующие интерфейс `Runable`.

Активный класс

Альтернативное обозначение активного класса (нотация UML 2.x)

Обозначения свойств классификатора с помощью теговых значений (1)

- *Теговое значение* (tagged value) - показывается на диаграмме с помощью текста в фигурных скобках:

  { имя_тега = значение_тега }

- Теговые значения – один из трех универсальных механизмов расширения языка UML. Этот механизм расширения применим к любому элементу модели, а не только к классификаторам.

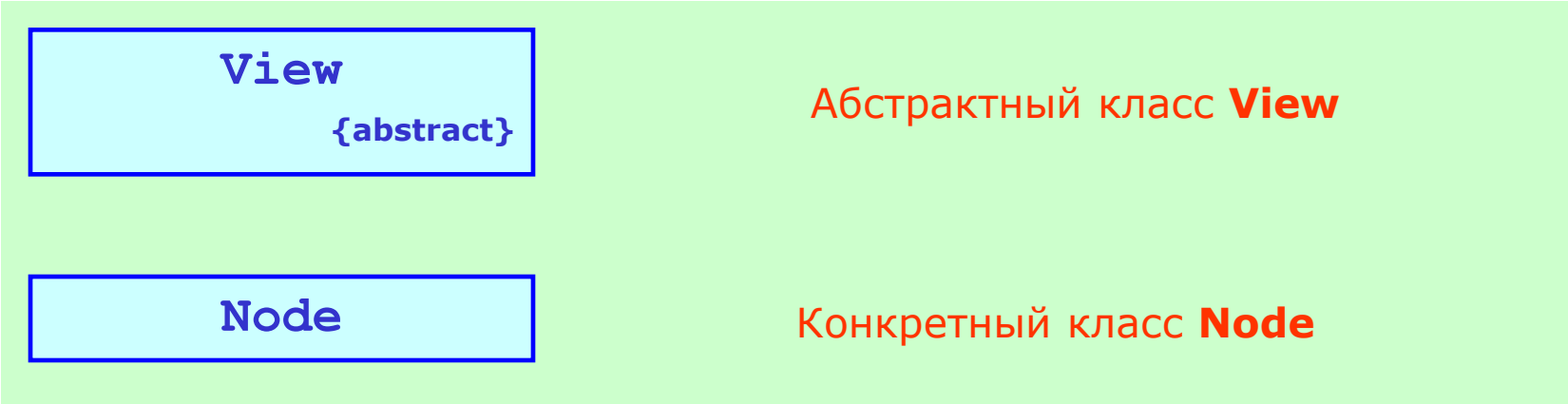
- Количество теговых значений, применимых к элементу модели, неограниченно. На диаграмме могут быть показаны не все теговые значения

- Теги для элемента могут быть заданы:
  - ✓ Определены в стандарте на язык UML
  - ✓Заданы пользователем при создании модели
  - ✓ Определены используемым CASE-инструментом

Обозначения свойств с помощью теговых значений (2)

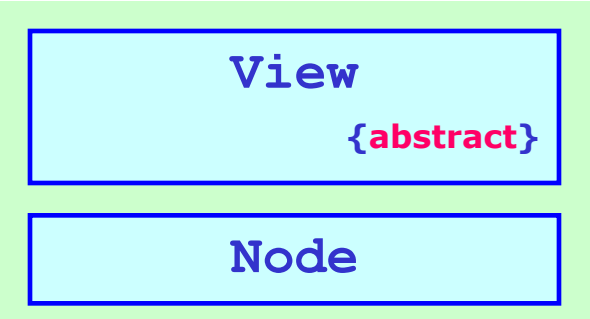
Для тегов принимающих логические значения **(true, false**) значение тега можно не указывать. Присутствие имени тега означает логическое значение **true**. Отсутствие имени тега означает логическое значение **false**.

Пример: представление свойства абстрактности не курсивом, а теговым значением.



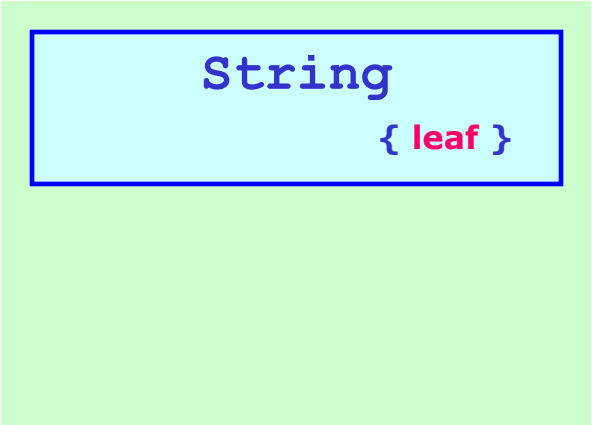
Реализация стандартных свойств языка UML: *abstract* и *leaf*

Реализация абстрактности класса (отсутствия экземпляров у класса).




```
// Java, C#
abstract class View {
}

class Node {
}
```

Реализация завершенности класса (**leaf** – лист в дереве наследования)
У завершенного класса не может быть классов-потомков.




```
// Java
final class String {
}
```

```
// C#
sealed class String {
}
```

Стереотипы классификаторов

- *Стереотип* – новый вид элемента модели, созданный на основе существующего. Стереотипы расширяют семантику модели.

- *Стереотип* – один их трех механизмов расширения языка UML

- Количество стереотипов у элемента модели (и у классификатора, в частности) неограниченно

- Стереотип характеризуется:
  - ✓ Расширяемым классификатором
  - ✓ Набором теговых значений
  - ✓ Набором ограничений (constraints)

Теги с произвольным типом значений

Edge
{author="Romanov"}

Тег ***author*** представляющий автора разработки класса в UML-модели и текстах реализации класса

Edge
{version="1.10.623"}

Тег ***version*** от системы управления версиями UML-модели

Edge
{file="Edge.java",
line=2,
column=8 }

Теги для связи UML-модели и текстов на языке программирования

```
/** Java
 * @author Romanov
 */
public class Edge {
}
```

```
/// C#
/// <author>Romanov</author>
///
public class Edge {
}
```

Изображение стереотипов классификаторов

Изображение стереотипа *boundary*

«boundary»
Edge

Текстовое изображение стереотипа для узла графа

Edge

Графическое изображение стереотипа для узла графа

Edge

Графическое изображение стереотипа для узла графа сжатого в пиктограмму

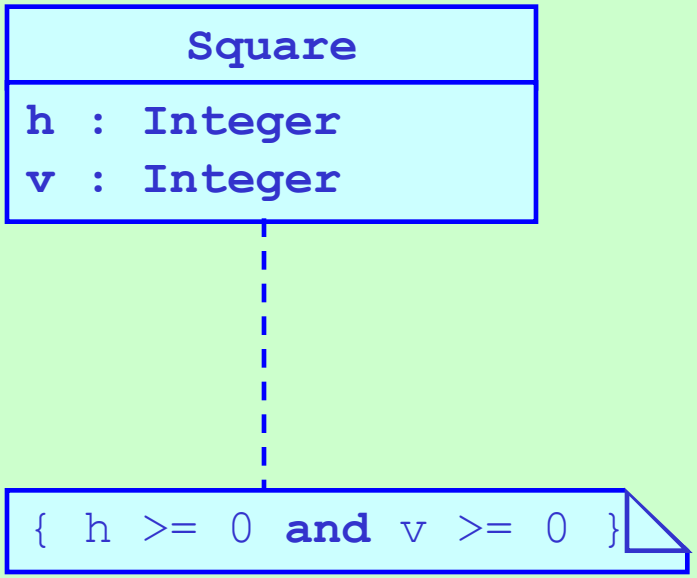
«control,singleton»
Sorter

Множество стереотипов класса *Sorter*

Ограничения накладываемые на UML-модель

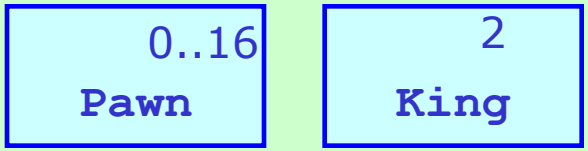
- *Ограничение (constraint)* – логическое выражение описанное с использованием атрибутов и отношений ассоциации классификаторов
- Истинно все время жизни элементов UML-модели
- Не изменяет состояние элементов UML-модели
- Может быть применено к любому элементу UML-модели
- Чаще всего ограничения описываются на языке **OCL** (**O**bject **C**onstraint **L**anguage)

Изображение ограничений на UML-диаграммах



Элемент модели, на который накладывается ограничение

Ограничение записанное на языке OCL и использующее атрибуты класса



Ограничение на количество экземпляров класса

Видимость классификаторов, их атрибутов и операций

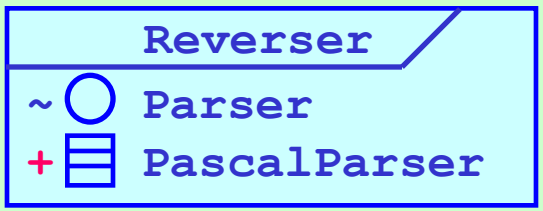
+ *публичная (public)* – элемент модели видим вне своего пространства имен

защищенная (protected) - элемент модели вне своего пространства имен видят только его потомки

— *скрытая (private)* - элемент модели невидим вне своего пространства

~ *пакетная (package)* - элемент модели невидим вне пакета содержащего классификатор владеющий данным элементом модели

Пример изображения видимости классификаторов (1)



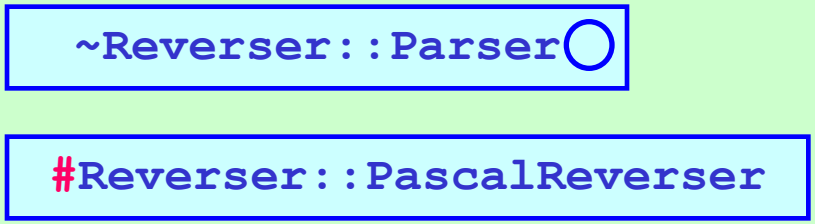

```java
// Java
package Reverser;

interface Parser {}
```

```java
// Java
package Reverser;

public class PascalParser {}
```

```csharp
// C#
namespace Reverser {
    interface Parser {
    }
}
```

```csharp
// C#
namespace Reverser {
    public class PascalParser {
    }
}
```

Пример изображения видимости классификаторов (2)




```
// Java
class Reverser {
    interface Parser {
    }
}
```

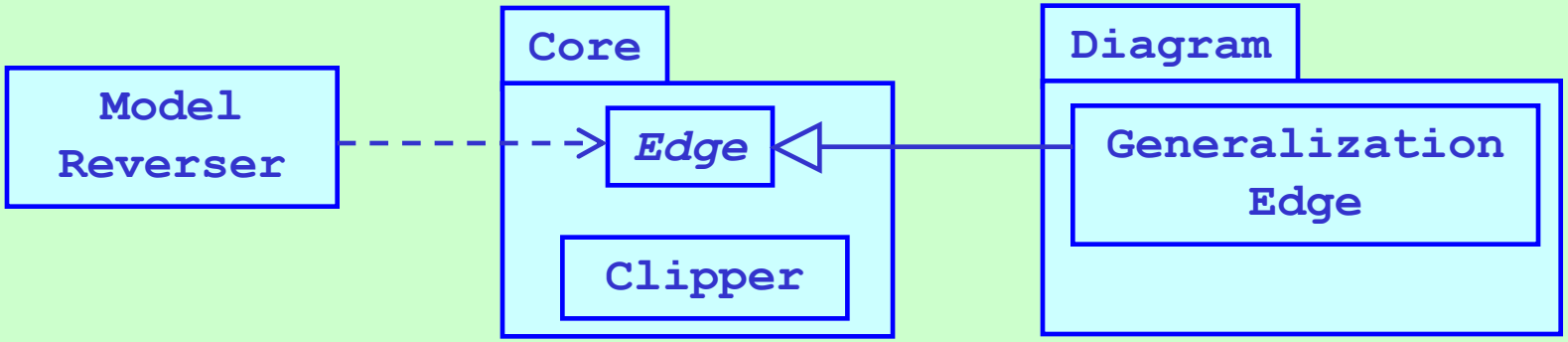
```
// Java
class Reverser {
 protected class PascalParser {
 }
}
```

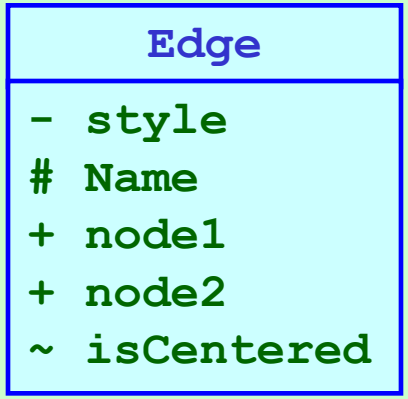
```
// C#
class Reverser {
    interface Parser {
    }
}
```

```
// C#
class Reverser {
  protected class PascalParser {
  }
}
```

Пример изображения видимости атрибутов и операций (1)

Атрибуты класса Edge видимые для класса Edge



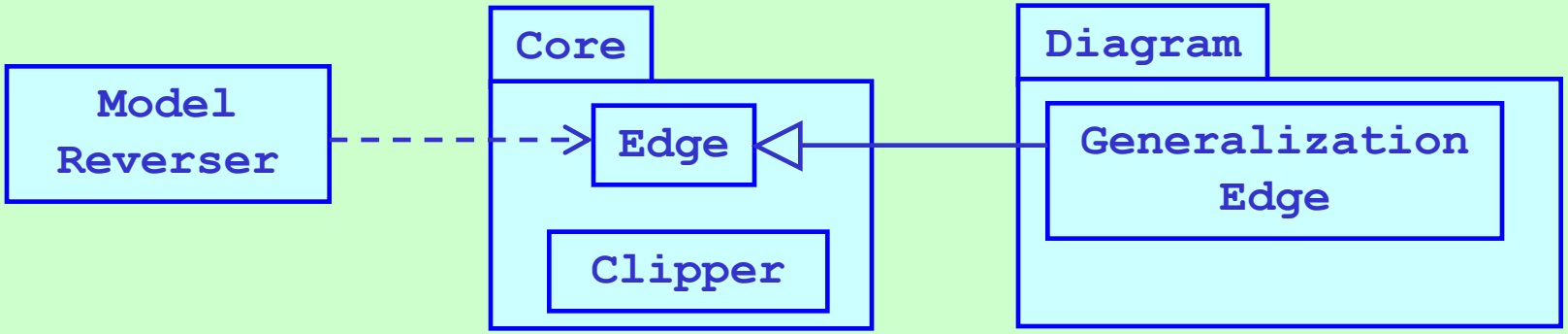


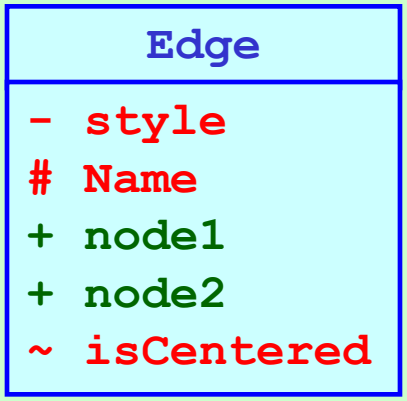
Зеленые – видимые элементы

Красные – не видимые элементы

Пример изображения видимости атрибутов и операций (2)

Атрибуты класса Edge видимые для класса ModelReverser



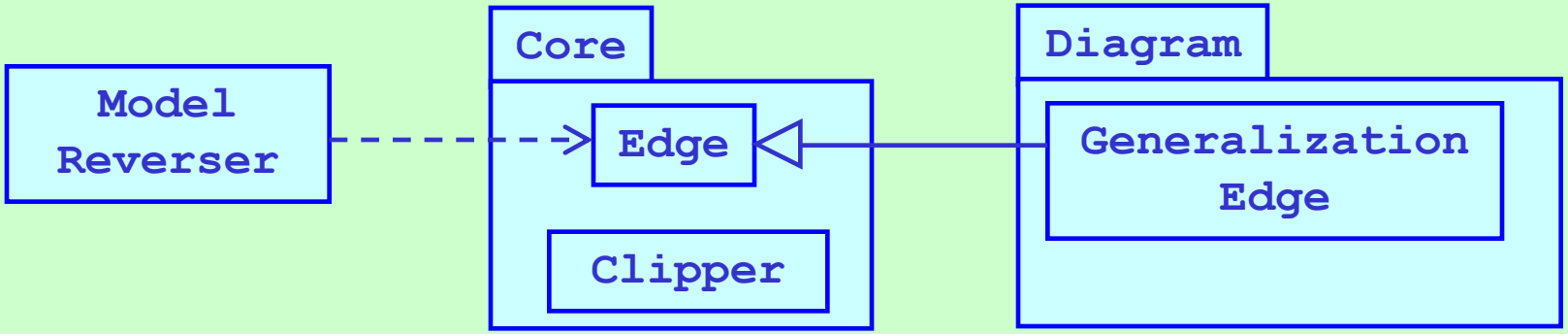


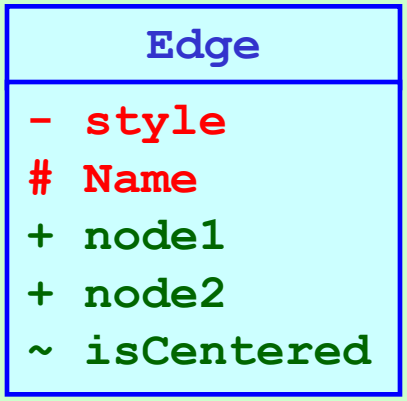
Зеленые – видимые элементы

Красные – не видимые элементы

Пример изображения видимости атрибутов и операций (3)

Атрибуты класса Edge видимые для класса Clipper



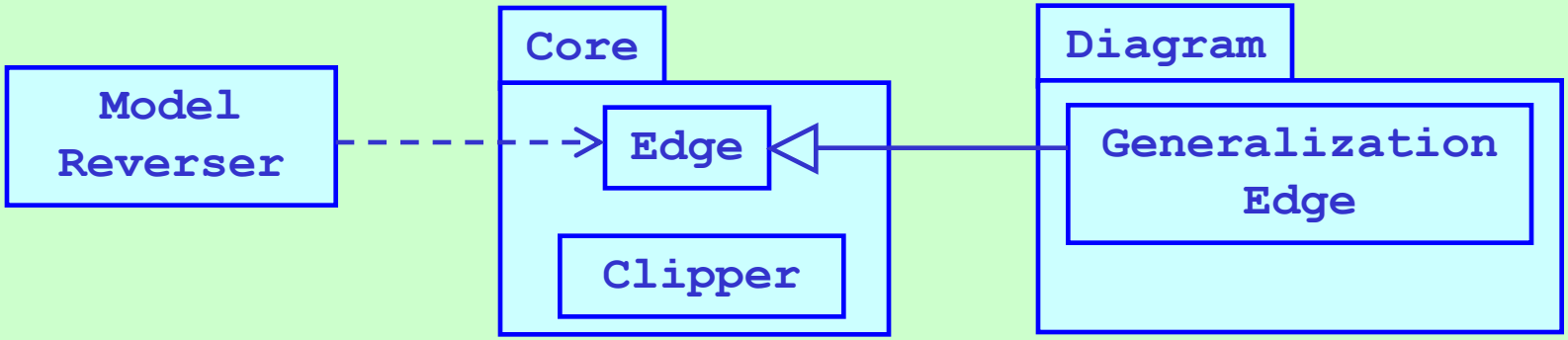


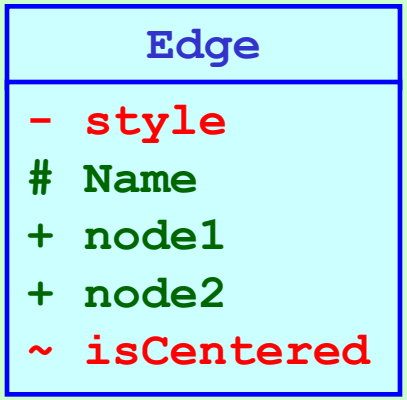
Зеленые – видимые элементы

Красные – не видимые элементы

Пример изображения видимости атрибутов и операций (4)

Атрибуты класса Edge видимые для класса GeneralizationEdge





Зеленые – видимые элементы

Красные – не видимые элементы

Область действия атрибутов и операций класса (1)

- Область действия – *класс*. Атрибуты и операции с такой область действия называются *статическими*. Они доступны всем экземплярам класса
- Область действия – *экземпляр класса*. Атрибуты и операции с такой область только этому экземпляру класса

Node
+ frameColor : Color + <u>defaultFrameColor: Color</u>

Статические элементы класса показываются подчеркиванием

Область действия атрибутов и операций класса (2)

Node
+ frameColor : Color + defaultFrameColor: Color

Статические элементы класса показываются подчеркиванием

```
// Java, C#
class Node {
   static public Color defaultFrameColor;
          public Color frameColor;
}
```

```
// C++
class Node {
public:
   static Color defaultFrameColor;
          Color frameColor;
};
```

Свойства атрибутов. Порожденные атрибуты (1)

Значения порожденных атрибутов не хранятся, а вычисляются в момент запроса.

Person
+ birthday : Date + /age : Integer

На диаграмме перед именем порожденного атрибута стоит символ **/**

```
// Java
class Person {
public Date birthday;

public int getAge() {
  return CurrentDate - birthday;
}
}
```

```
// C#
class Person {
public Date birthday;

public int age {
  get { return CurrentDate -
                birthday; }
}
}
```

Свойства атрибутов. Порожденные атрибуты (2)

Значения порожденных атрибутов не хранятся, а вычисляются в момент запроса.

«interface» IPerson
+ /age : Integer

На диаграмме перед именем порожденного атрибута стоит символ **/**

```
// Java
interface Person {
    public int getAge();
}
```

```
// C#
interface Person {
   public int age { get; }
}
```

Свойства атрибутов. Неизменяемые атрибуты

«utility» Math
+ pi : Double { frosen }

Атрибуты *frosen* вычисляются на этапе компиляции.

```
// Java
class Math {
public final double pi = 3.14;
}
```

```
// C#
class Math {
public const double pi = 3.14;
}
```

```
// C++
class Math {
public:
    const pi = 3.14;
}
```

Свойства атрибутов. Атрибуты только для чтения

History
+ events : List { readonly }

Атрибуты *readonly* неизменяемы вне класса. Внутри класса возможны чтение и запись.

```
// C#
class Math {
  public readonly List events;
}
```

Свойства методов. Методы - запросы

Node
+ paint(g : Graphics){ query }

```
// C++
class Node {
public:
  void paint(Graphics g) const;
}
```

1. Метод-запрос не может изменять унаследованные и собственные атрибуты
2. Метод-запрос не может не может вызывать собственные и унаследованные методы, которые не являются методами-запросами.

Свойства методов. Не переопределяемые методы

Account
+ Withdraw(n : long) {leaf}

leaf – запрет переопределения метода в классах-потомках.

```
// C#
class Account {
  leaf
  public void Withdraw(long n)
  {
   …
  }
}
```

```
// Java
class Account {
  final
  public void Withdraw(long n)
  {
   …
  }
}
```

Свойства методов. Не переопределяемые методы

Account
+ Withdraw(n : long) {leaf}

leaf – запрет переопределения метода в классах-потомках.

```
// C#
class Account {
  leaf
  public void Withdraw(long n);
}
```

```
// Java
class Account {
  final
  public void Withdraw(long n);
}
```

Свойства методов. Абстрактность методов

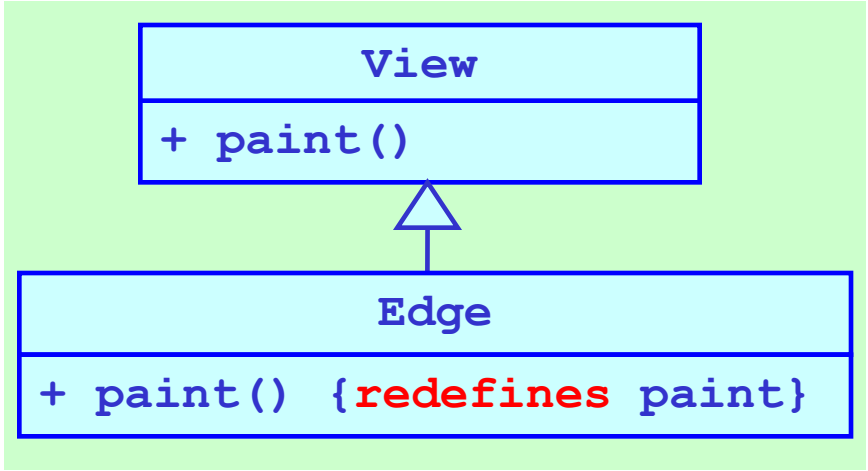
View
+ paint(g : Graphics) {abstract}

abstract – отсутствие реализации у метода.
Абстрактность может показываться также курсивом.

```
// C#, Java
abstract class View {
  abstract public void paint(Graphics g);
}
```

```
// C++
class View {
public:
  virtual void paint(Graphics g) = 0;
}
```

Свойства методов. Переопределение методов

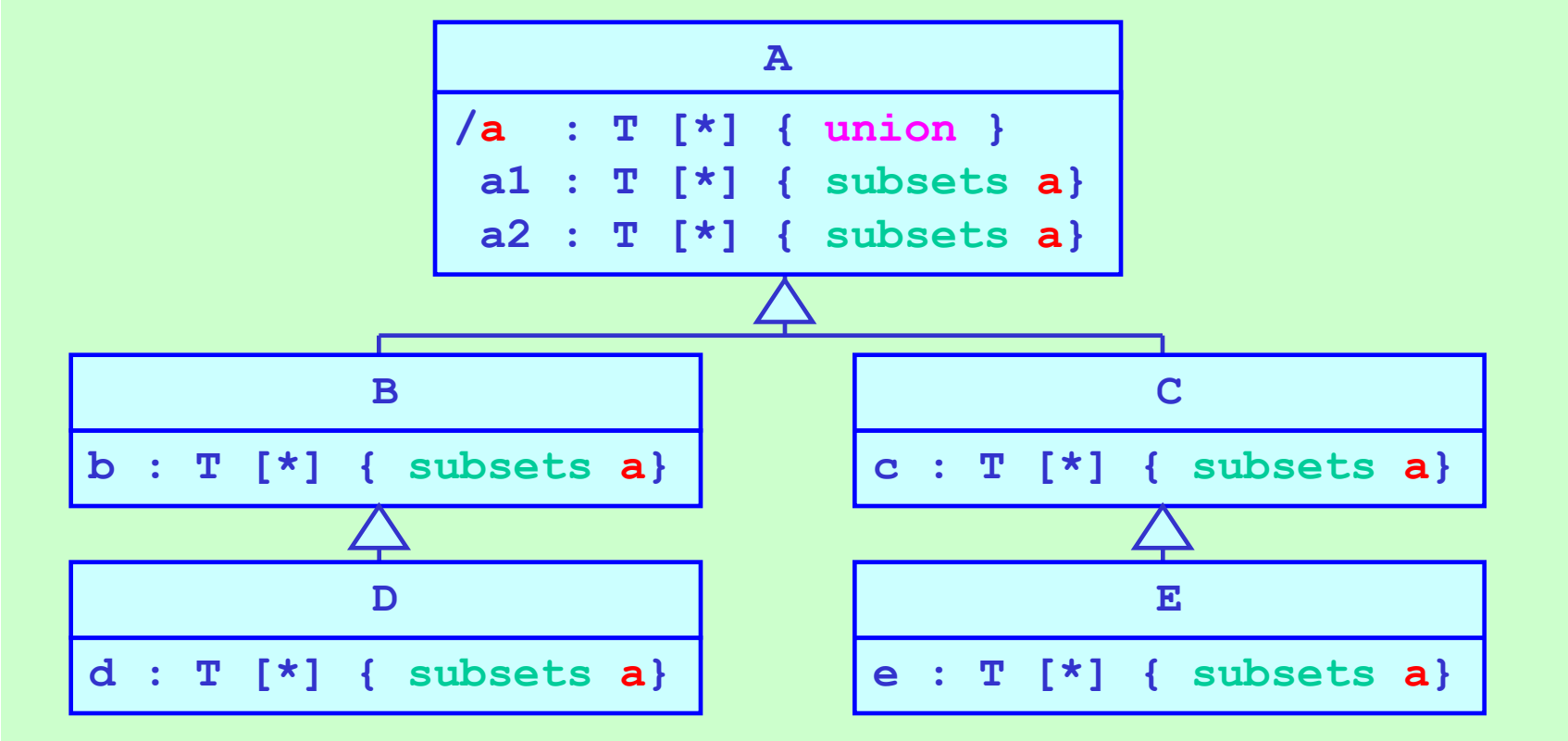



```java
// Java
class View {
   public void paint();
}


class Edge extends View {
  public void paint();
}
```

```csharp
// C#
class View {
   virtual public void paint();
}

class Edge : View {
   override public void paint();
}
```

```cpp
// C++
class View {
public:
  virtual void paint();
};

class Edge : View {
public:
  void paint();
};
```

Объединения и подмножества атрибутов




```
// Java
D x = new D();
T[] y = x.getA();
// y = a1 + a2 + b + d
```

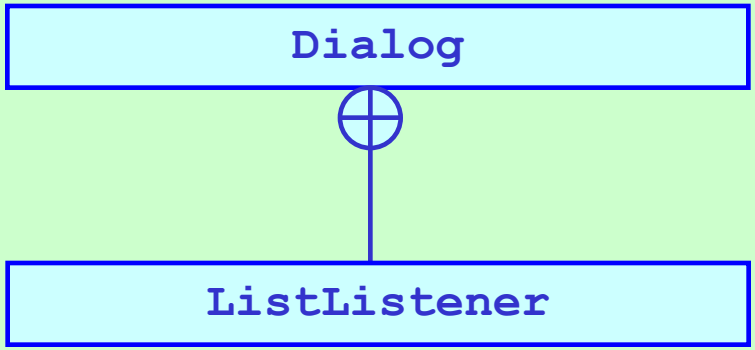
```
// Java
B x = new B();
T[] y = x.getA();
// y = a1 + a2 + b
```

```
// Java
A x = new A();
T[] y = x.getA();
// y = a1 + a2
```

Вложенность классификаторов

Вложенность классификатора *ListListener* показывается в отдельной секции классификатора *Dialog*.

Вложенность классификатора *ListListener* показывается с помощью квалифицированного имени.



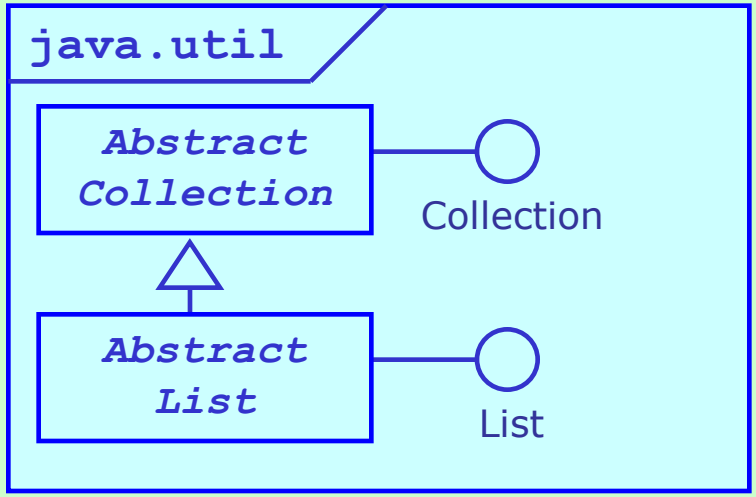
Вложенность классификатора *ListListner* показывается с помощью отношения вложенности между узлами графа *ListListener* и *Dialog*.

1.3 Классификаторы и пакеты в нотации языка UML

- Классификаторы
- Свойства классификаторов
- Структурирование модели с помощью пакетов
- Шаблоны для классификаторов и функций

Изображение пакетов

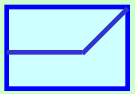
Изображение «раскрытого» пакета java.util с вложенными в пакет классами и интерфейсами



Изображение «раскрытого» пакета java.util с вложенной в пакет UML-диаграммой, показывающей отношения между вложенными в пакет классами.

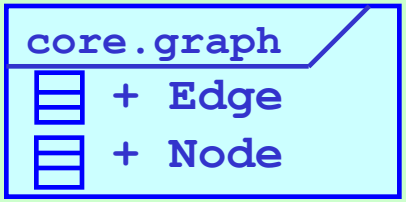
Представление пакетов в языках C++, Java и C#

Изображение «сжатого» пакета java.util



java.util

Изображение-пиктограмма пакета java.util




```
// C#
namespace core.graph {
    class Edge {};
    class Node {};
}
```

```
// Java
package core.graph;

public class Edge {
}
```

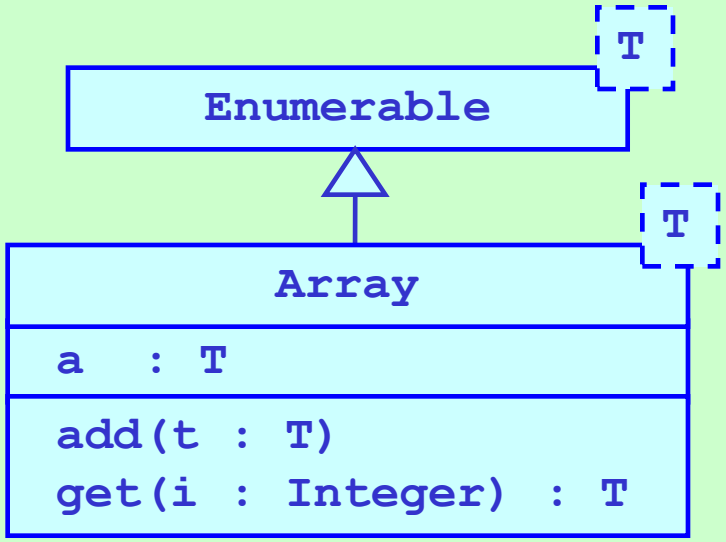
```
// Java
package core.graph;

public class Node {
}
```

```
// C++
namespace core {
    namespace graph {
        class Edge {};
        class Node {};
    }
}
```

1.4 Шаблоны для классификаторов и функций

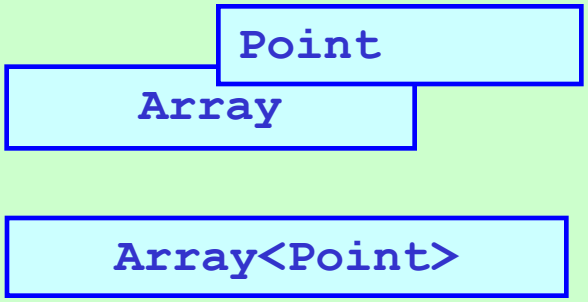
- Классификаторы
- Свойства классификаторов
- Пакеты и их свойства
- Шаблоны для классификаторов и функций

Шаблоны для классификаторов и функций. Шаблон класса C#



Объявление шаблона класса

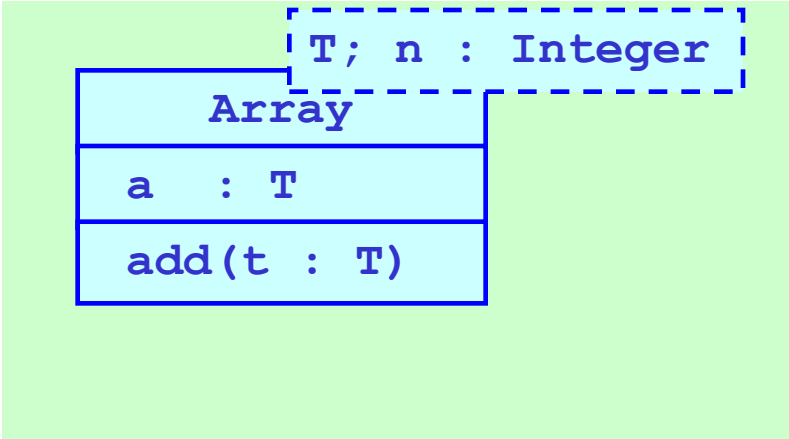
```
// C# 2.0
class Array<T> : Enumerable<T> {
   public T a;
   public void add(T t) {}
   public T get(int i) {}
}
```



Настройка шаблона класса

```
// C# 2.0
Array<Point> path;
```

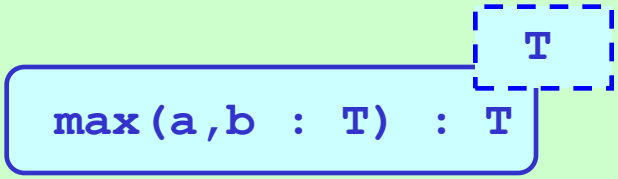
Шаблоны для классификаторов и функций. Шаблон класса C++




```
// C++
template <class T, int n>
class Array {
public:
   T a;
   void add(T t);
};
```

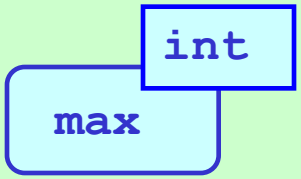

```
// C++
Array<Point, 100> path;
```

Шаблоны для классификаторов и функций. Шаблон функции C++



Объявление шаблона функции

```
// C++
template <class T>
T max(T a, T b)
   { return a > b ? a : b; }
```



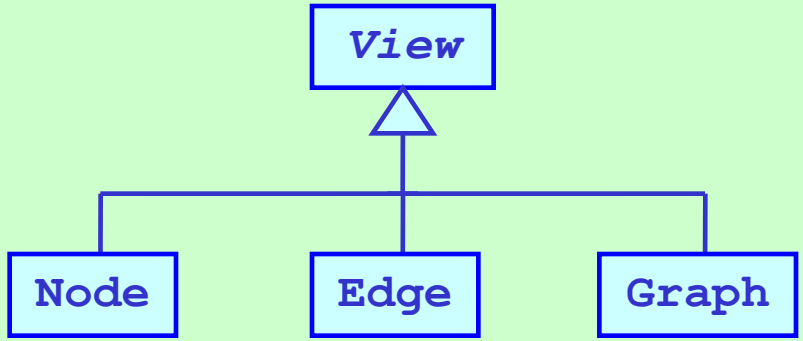
Настройка шаблона функции

```
// C++
int x = max<int>(y, 2);
```

2.1 Отношение обобщения

- Отношения обобщения
- Отношения реализации
- Отношение ассоциации
- Отношение зависимости

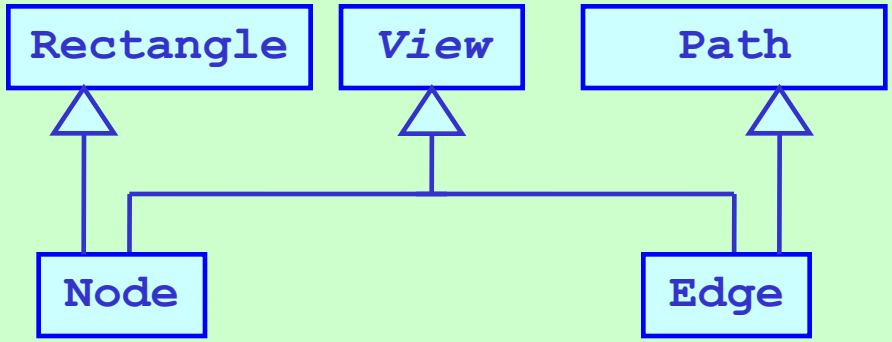
Отношения обобщения.
Особенности языков C++, Java и C#



Одиночное наследование в Java и C#

```
// Java
class Node extends View {
}
```

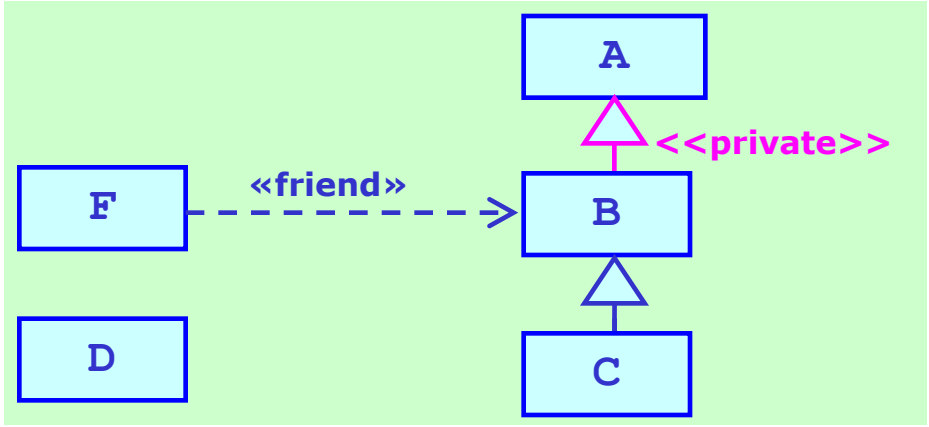
```
// C#
class Node : View {
}
```



Множественное наследование в C++

```
// C++
class Node : public View,
             public Rectangle
{
};

class Edge : public View,
             public Path

{
};
```

Отношения обобщения.
Скрытое наследование в языке C++




```
// C++
class A {
  private:   int x;
  protected: int y;
  public:    int z;
};
```

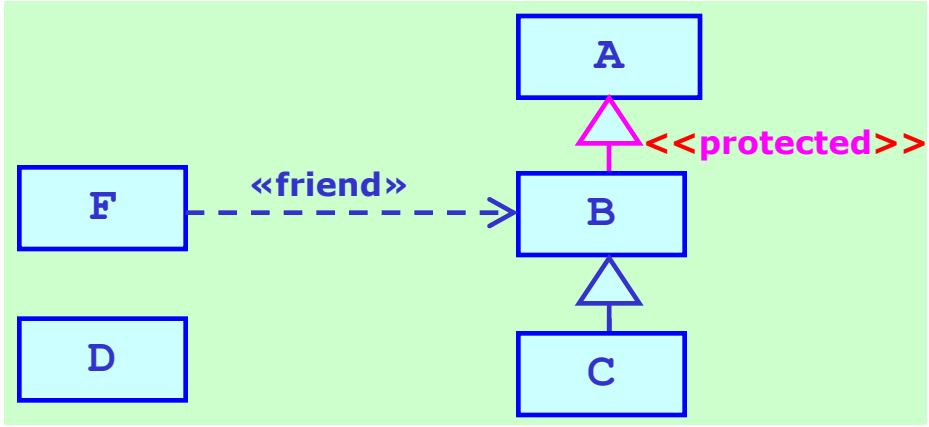
```
class B : private A {
  friend F;

  void aMember() {
     x = 1; // не допустимо
     y = 1; // допустимо
     z = 1; // допустимо
}};
```

```
class C : public B {
  void aMember() {
     x = 1; // не допустимо
     y = 1; // не допустимо
     z = 1; // не допустимо
}};
```

```
class D {
  A a;
  void aMember() {
     a.x = 1; // не допустимо
     a.y = 1; // не допустимо
     a.z = 1; // допустимо
}};
```

```
class F { //
  A a;
  void aMember() {
     a.x = 1; // не допустимо
     a.y = 1; // не допустимо
     a.z = 1; // допустимо
}};
```

Отношения обобщения.
Защищенное наследование в языке C++




```
// C++
class A {
  private:   int x;
  protected: int y;
  public:    int z;
};
```

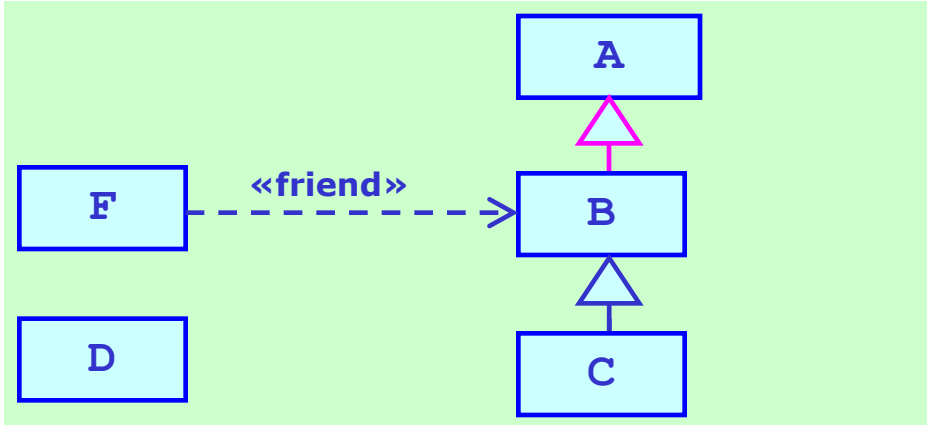
```
class B : protected A {
  friend F;

  void aMember() {
     x = 1; // недопустимо
     y = 1; // допустимо
     z = 1; // допустимо
}};
```

```
class C : public B {
  void aMember() {
     x = 1; // недопустимо
     y = 1; // допустимо
     z = 1; // допустимо
}};
```

```
class D {
  void aMember() {
     x = 1; // недопустимо
     y = 1; // допустимо
     z = 1; // допустимо
}};
```

```
class F { //
  void aMember() {
     x = 1; // недопустимо
     y = 1; // допустимо
     z = 1; // допустимо
}};
```

Отношения обобщения.
Открытое наследование в языке C++




```
// C++
class A {
  private:   int x;
  protected: int y;
  public:    int z;
};
```

```
class B : public A {
  friend F;

  void aMember() {
     x = 1; // недопустимо
     y = 1; // допустимо
     z = 1; // допустимо
}};
```

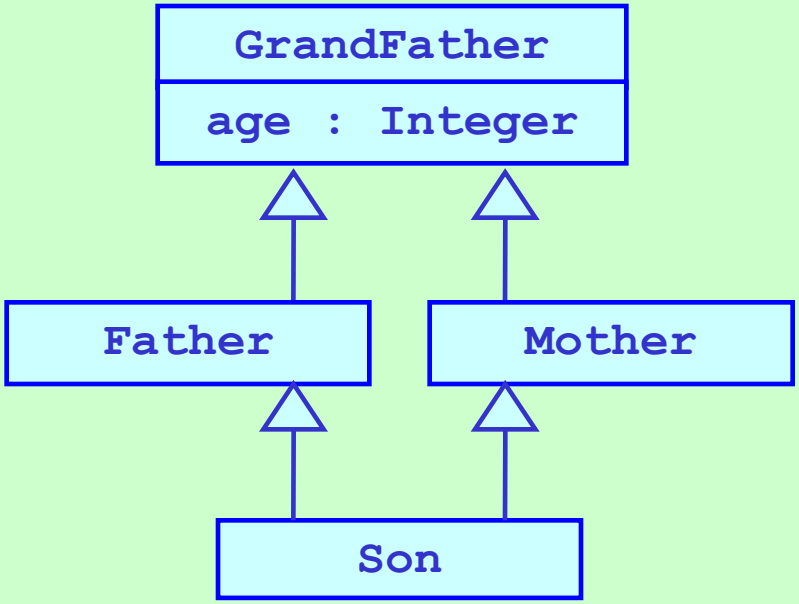
```
class C : public B {
  void aMember() {
     x = 1; // недопустимо
     y = 1; // допустимо
     z = 1; // допустимо
}};
```

```
class D {
  void aMember() {
     x = 1; // недопустимо
     y = 1; // допустимо
     z = 1; // допустимо
}};
```

```
class F { //
  void aMember() {
     x = 1; // недопустимо
     y = 1; // допустимо
     z = 1; // допустимо
}};
```

Отношения обобщения.

Множественное наследование в языке C++



Наследование атрибута *age* по материнской и отцовской линии!

Происходит дублирование атрибута *age* в классе-потомке!

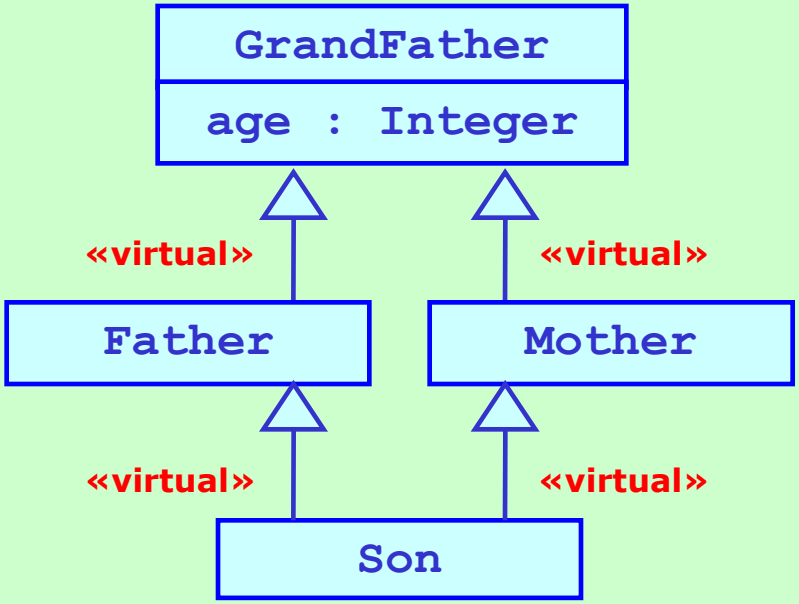
Не всегда дублирование атрибута приемлемо для модели!

```
class Son :
  public Mother, public Father {
  void aMember() {
     age = 1; // недопустимо
  }
};
```

```
class Son :
  public Mother, public Father
  void aMember() {
     Father::age = 1; // Допустимо
     Mather::age = 1; // Допустимо
  }
};
```

Отношения обобщения.

Виртуальное наследование в языке C++



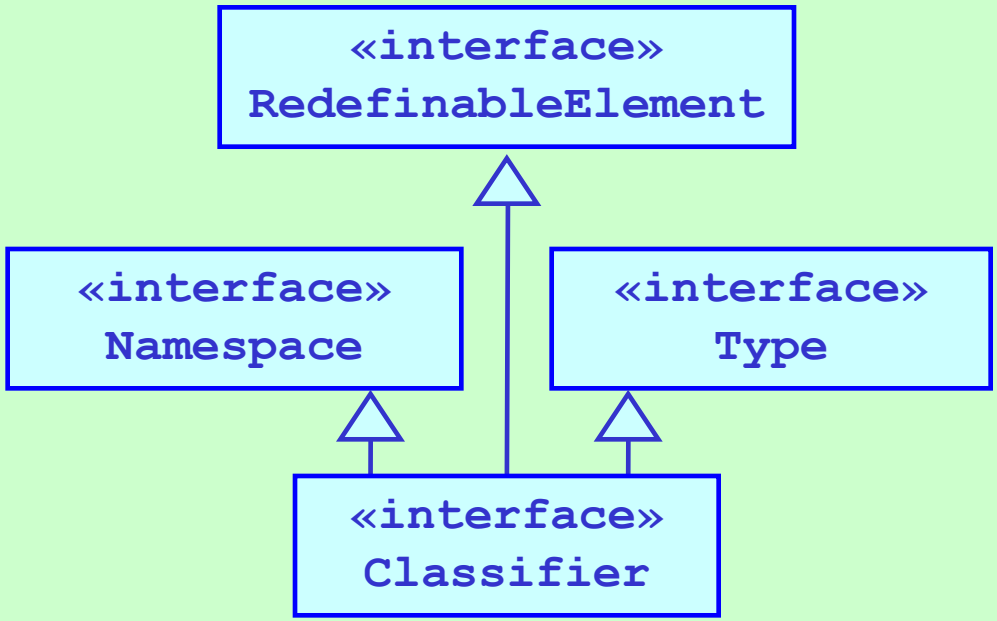
При виртуальном наследовании дублирования атрибута *age* не происходит!

```
class Son :
   virtual public Father,
   virtual public Mother {
  void aMember() {
     age = 1; // Допустимо
  }
};
```

```
class Father : virtual public GrandFather {
};
```

```
class Mother : virtual public GrandFather {
};
```

Отношения обобщения.
Наследование между интерфейсами



Множественное наследование допустимо между интерфейсами в языках Java и C#

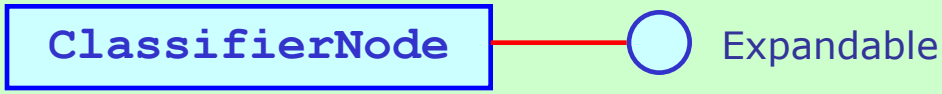
```
// Java
interface Classifier
extends Namespace,
        RedefinableElement,
        Type
{
}
```

```
// C#
interface Classifier
: Namespace,
  RedefinableElement,
  Type
{
}
```

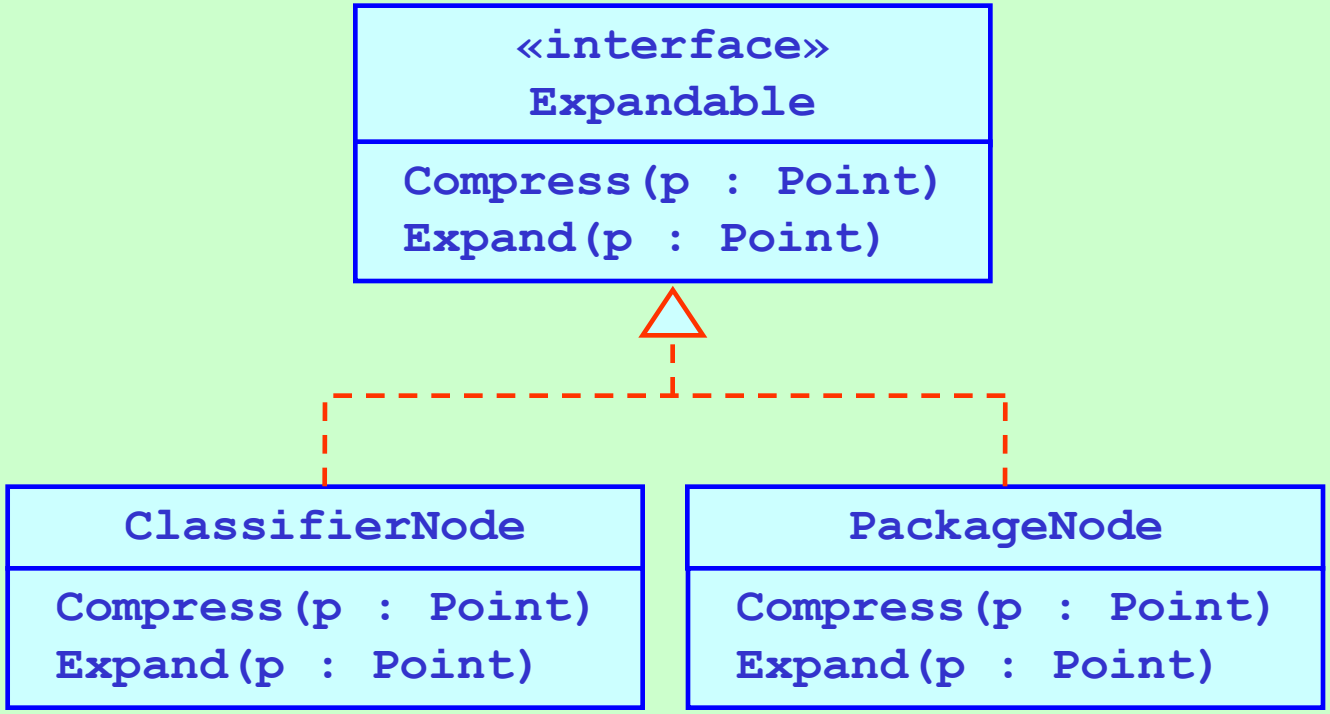
2.2. Отношение реализации

- Отношения обобщения
- Отношения реализации
- Отношение ассоциации
- Отношение зависимости

Отношения реализации.
Варианты изображения отношения



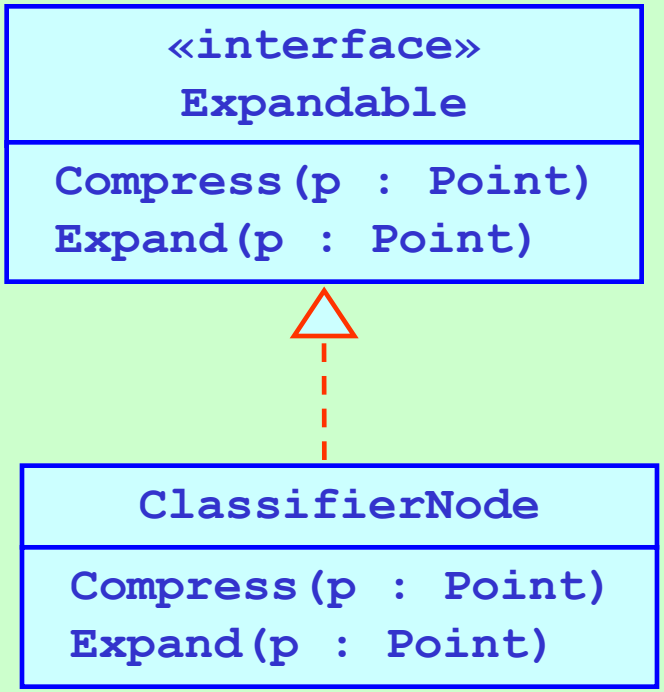
Отношение реализации интерфейса показанного в сжатом состоянии



Отношение реализации интерфейса показанного в раскрытом состоянии

Отношения реализации.

Реализация интерфейса в языках Java и C#



Реализация операций интерфейса возможна в классе ClassifierNode или в его потомках

```
// Java
interface Expandable {
   void Compress(Point p);
   void Expand(Point p);
}
class ClassifierNode
implements Expandable {
   void Compress(Point p) {…}
   void Expand(Point p)   {…}
}
```

```
// C#
interface Expandable {
   void Compress(Point p);
   void Expand(Point p);
}
class ClassifierNode
: Expandable {
   void Compress(Point p) {…}
   void Expand(Point p)   {…}
}
```

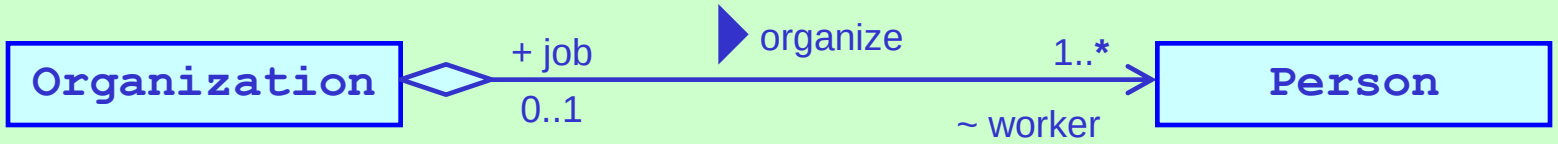
2.3. Отношение ассоциации

- Отношения обобщения
- Отношения реализации
- Отношение ассоциации
- Отношение зависимости

Отношение ассоциации.

Изображение ассоциации и ее свойств

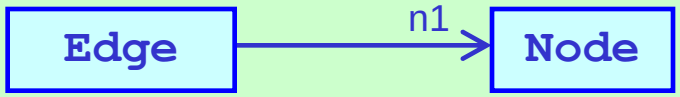
Отношение ассоциации описывает взаимосвязь между экземплярами классификаторов



organize	Имя ассоциации	
▶	Направление чтения ассоциации	
job	Имя окончания ассоциации	worker
0..1	Множественность окончания ассоциации	1..*
+	Видимость окончания ассоциации	~
◇	Свойство агрегации окончания ассоциации	
	Свойство навигации окончания ассоциации	>

Отношение ассоциации.

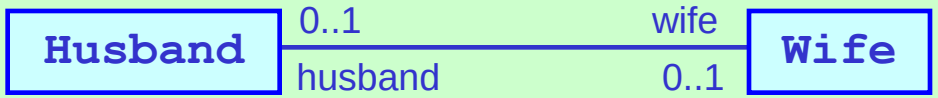
Свойство навигации у окончания ассоциации



Навигация означает возможность одного экземпляра класса ссылаться на другой экземпляр класса.

Навигация показывается стрелкой на окончании ассоциации.

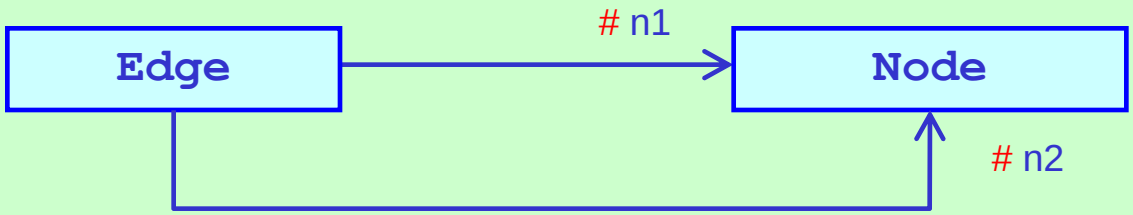
```
// C++
class Node {
};

class Edge {
public:
   Node *n1;
};
```



Отсутствие стрелок означает возможность навигации по обоим направлениям.

```
// C++
class Husband {
public:
   Wife *wife;
};

class Wife {
public:
   Husband *husband;
};
```

Отношение ассоциации.
Свойство видимости окончания ассоциации



Разновидности видимости

+ открытая - скрытая # защищенная ~ пакетная

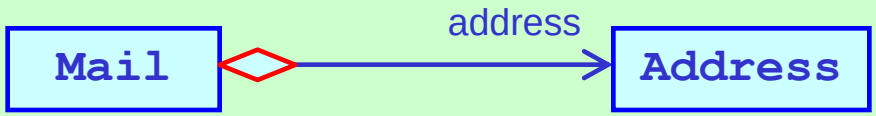
```
// Java
public class Node {
}
```

```
// Java
public class Edge {
   protected Node n1;
   protected Node n2;
}
```

```
// C#
public class Node {
}

public class Edge {
   protected Node n1;
   protected Node n2;
}
```

```
// C++
class Node {
};

class Edge {
protected:
   Node *n1;
   Node *n2;
};
```

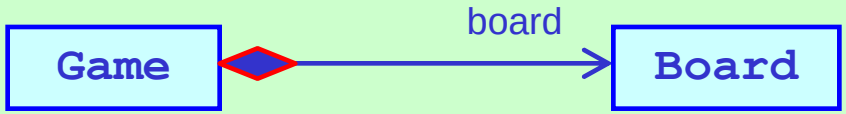
Отношение ассоциации.
Свойство агрегации и композиции



Агрегация описывает отношение часть-целое.

Время жизни части и целого могут не совпадать.

```
// C#
public class Address {
}

public class Mail {
   public Address address;
}
```

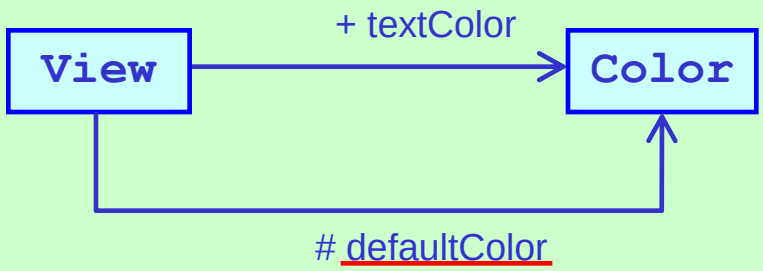


Композиция означает владение одного экземпляра класса другим экземпляром класса.

Время жизни собственника и собственности совпадают.

```
// C++
class Game {
public:
   Board *board;
   Game()
    { board = new Board(); }
   ~Game()
     { delete board; }
};
```

Отношение ассоциации.
Область действия (scope) окончания ассоциации



Если область действия окончания ассоциации – класс, то оно доступно всем экземплярам класса.

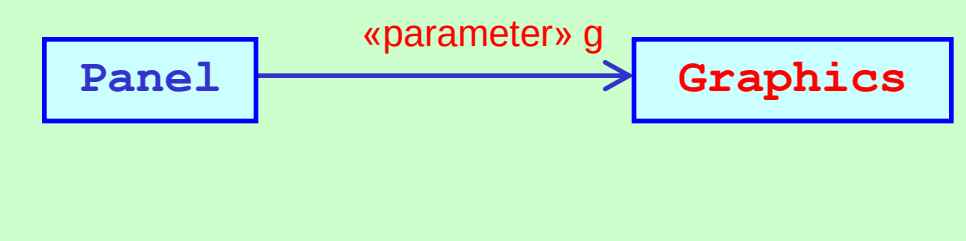
Окончания с областью действия – класс подчеркнуты.

```
// C#
public class View {
   public Color textColor;

   static
   protected Color defaultColor;
}
```

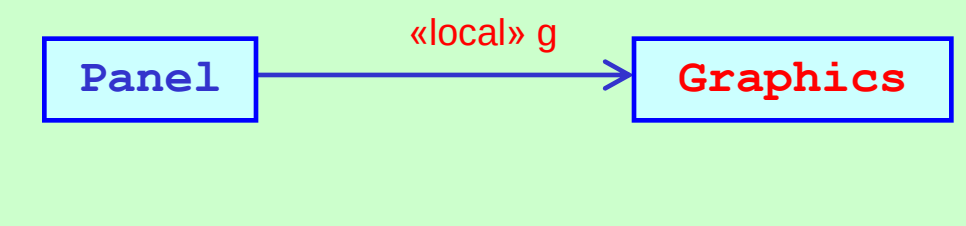
```
// Java
public class View {
   public Color textColor;

   static
   protected Color defaultColor;
}
```

```
// C++
class View {
public:
   Color *textColor;

protected:
   static
   Color *defaultColor;
};
```

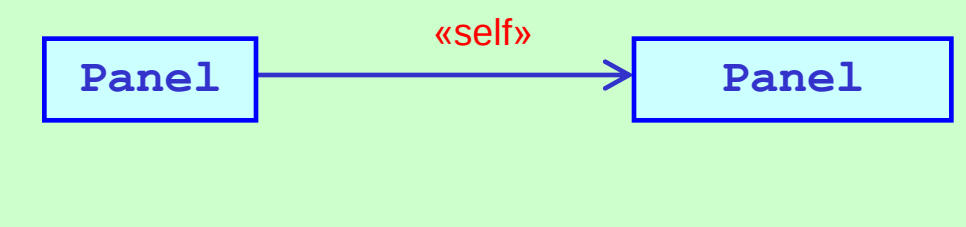
Отношение ассоциации.
Стереотипы окончания ассоциации



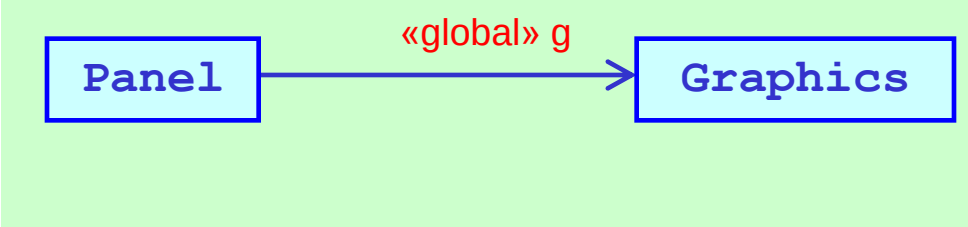

```
// C#
public class Panel {
   public void draw(Graphics g)
   {…}
}
```




```
// C#
public class Panel {
   public void draw()
   { Graphics g; }
}
```

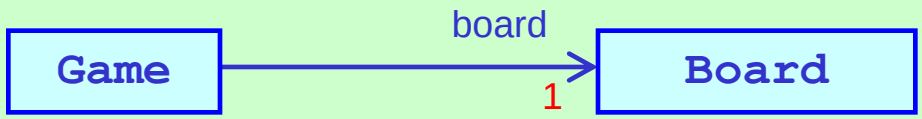



```
// C#
public class Panel {
   public void draw()
   { this.expand(); }
}
```

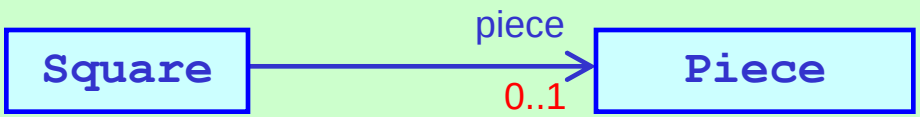



```
// C++
Graphics g;
class Panel {
   public void draw()
      { g.update(); }
};
```

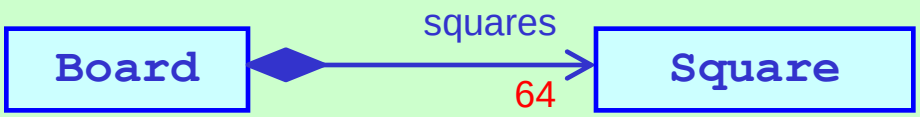
Отношение ассоциации.
Множественность окончания ассоциации (1)




```
// C#
public class Game {
   public Board board;
   public Game(Board b)
      { board = b; }
}
```

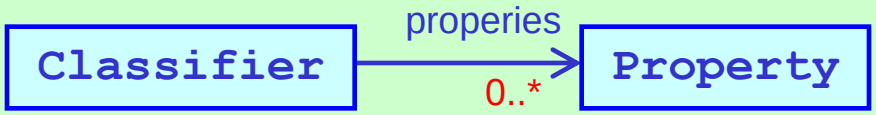



```
// C#
public class Square {
   public Piece piece;
   public Square()
       { }
}
```

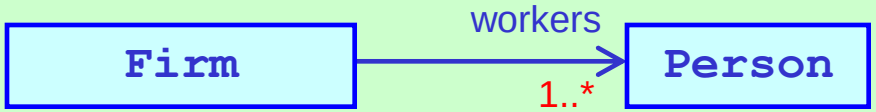



```
// C#
public class Board {
   public Square[] squares;
   public Board()
   { squares = new Square[64]; }
}
```

Отношение ассоциации.
Множественность окончания ассоциации (2)




```
// C#
using System.Collections.Generic;
public class Classifier {
  public ISet<Property> properties;
  public Classifier() {
    properties = new HashSet<Property>();
}}
```

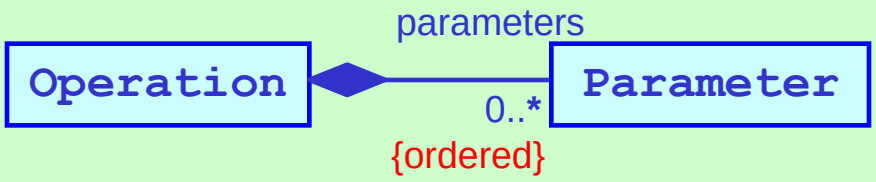



```
// C#
using System.Collections.Generic;
public class Firm {
   public ISet<Person> workers;
   public Firm(Person chief) {
      workers = new HashSet<Person>();
      workers.Add(chief);
}}
```

```
// C++
#include <set>
using std::set;

class Firm {
public:
   set<Person*> workers;
   Firm(Person *chief) {
       workers.insert(chief);
}};
```

```
// Java
import java.util.*;

public class Firm {
   public Set<Person> workers;
   public Firm(Person chief) {
      workers = new HashSet<Person>();
      workers.add(chief);
   }
}
```

Отношение ассоциации.
Упорядоченность окончания ассоциации



Умалчиваемое значение свойства {unordered}

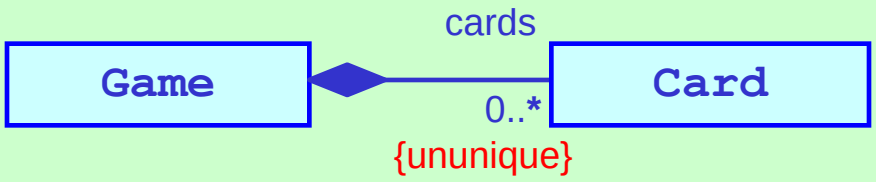
```
// C#
using System.Collections.Generic;
public class Operation {
 public IList<Parameter> parameters;
 public Operation() {
   parameters = new List<Parameter>();
 }
 public void addParameter(Parameter p) {
    if (parameters.Contains(p)) return;
    parameters.Add(p);
 }
}
```

```
// Java
public class Operation {
 public List<Parameter> parameters;

 public Operation() {
    parameters =
       new ArrayList<Parameter>();
 }

 public void addParameter(Parameter p) {
    if (parameters.Contains(p)) return;
    parameters.Add(p);
 }
}
```

```
// C++
#include <vector>
#include <algorithm>
using namespace std;

class Operation {
public:
   vector<Parameter> ps;
   void addParameter(Parameter *p);
};
void Operation::addParameter(Parameter *p) {
  if (find(ps.begin(), ps.end(), p) != ps.end())
     return;
  ps.push_back(p);
}
```

Отношение ассоциации.

Уникальность окончания ассоциации



Умалчиваемое значение свойства {ununique}

```
using System.Collections.Generic;

public class Game {
    public IList<Card> cards;
    public Game() {
        cards = new List<Card>();
    }
}
```

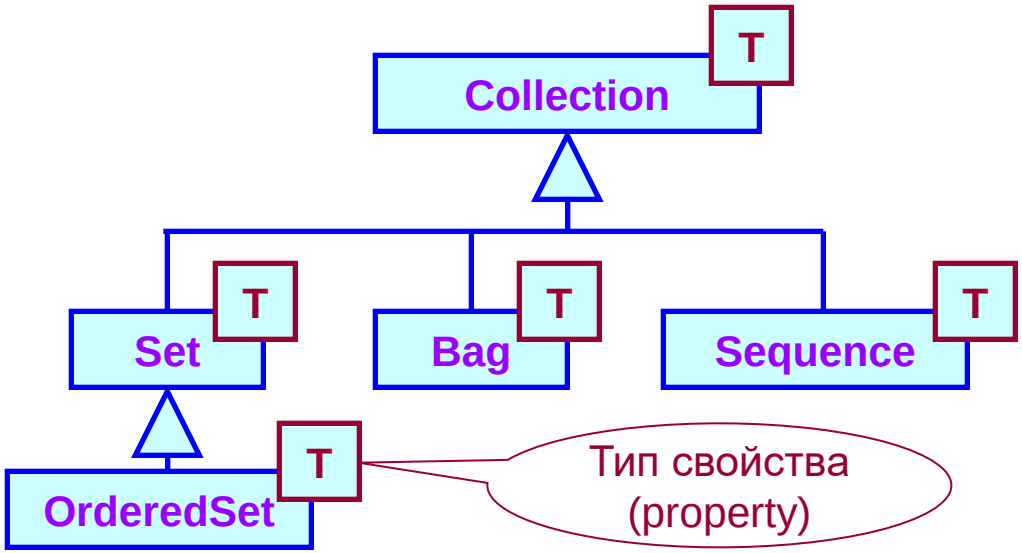
```
// Java
import java.util.*;

public class Game {
   public List<Card> cards;
   public Game() {
       parameters =
            new ArrayList<Card>();
   }
}
```

```
// C++
#include <vector>
using namespace std;

class Game {
  public:
     vector<Card> cards;
};
```

Отношение ассоциации.
Уникальность, упорядоченность и язык OCL

OCL коллекции

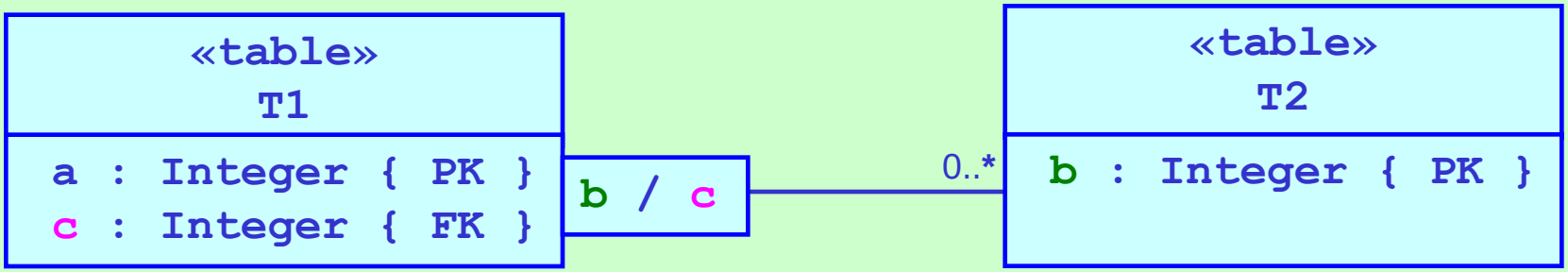
	{unique}	{ununique}
{unordered}	**Set**	**Bag**
{ordered}	**OrderedSet**	**Sequence**



Отношение ассоциации.

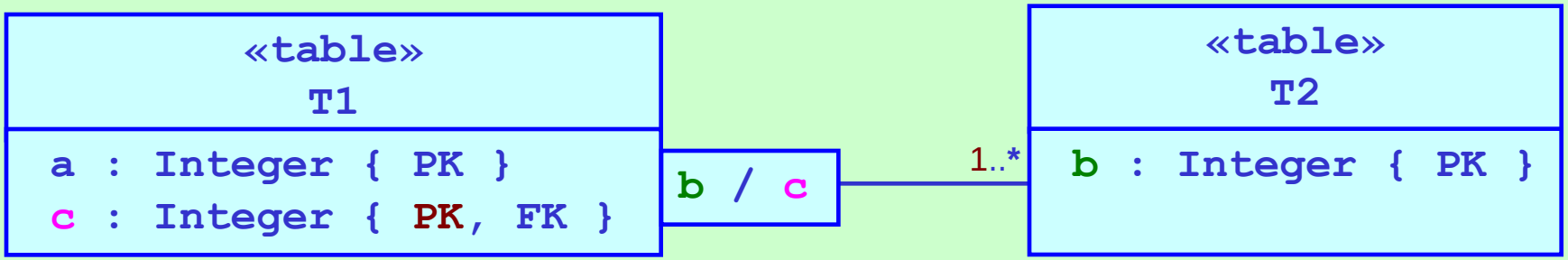
Квалификатор окончания ассоциации

Описание миграции первичных ключей и превращения их во внешние ключи



b – первичный ключ {PK} таблицы T2 мигрирует в таблицу T1
и становится внешним ключом {FK} таблицы T1 с именем **c**.

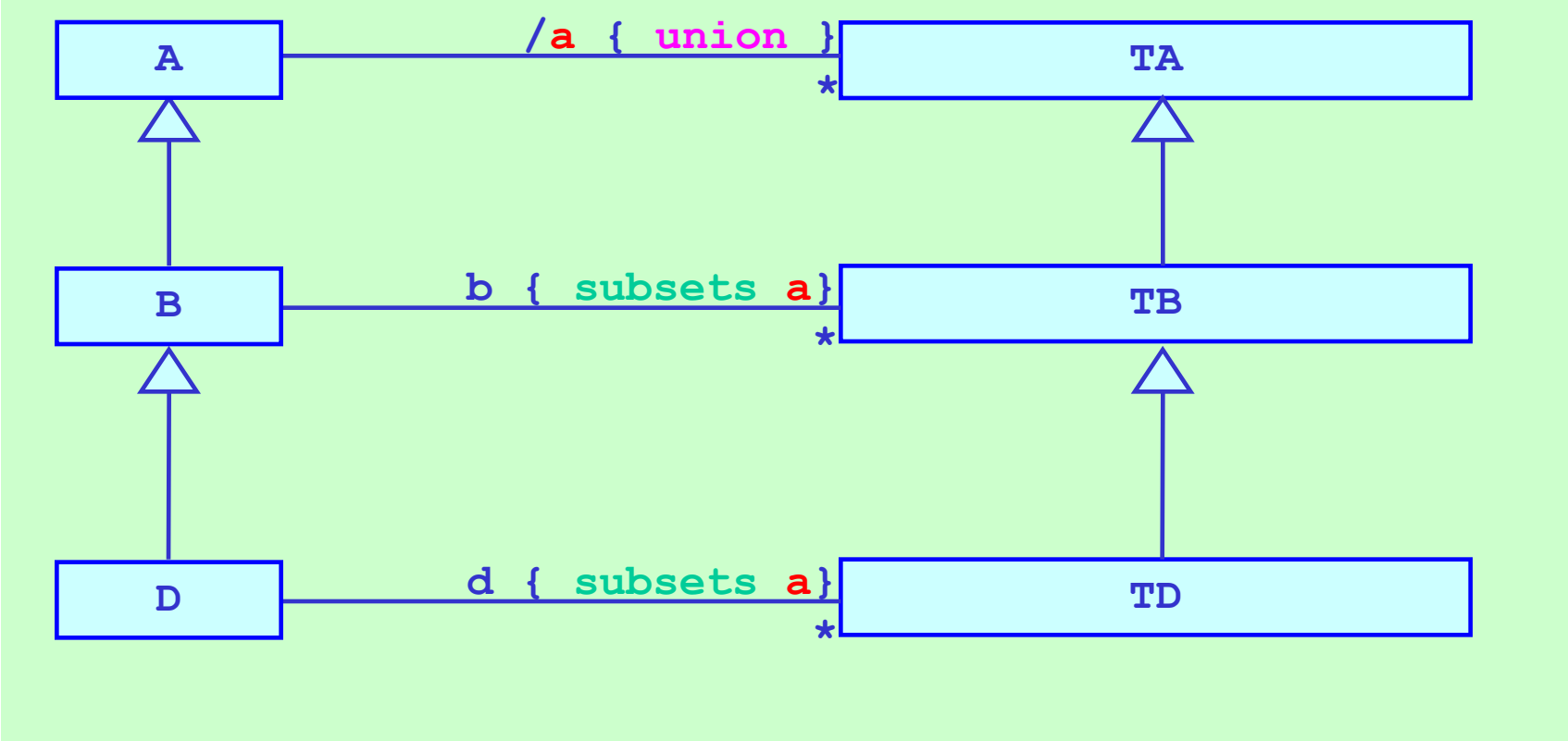
Описание миграции первичных ключей и превращения их в первичные внешние ключи



b – первичный ключ {PK} таблицы T2 мигрирует в таблицу T1
и становится первичным внешним ключом {FK} таблицы T1 с именем **c**.

Отношение ассоциации.

Объединения и подмножества в ассоциации



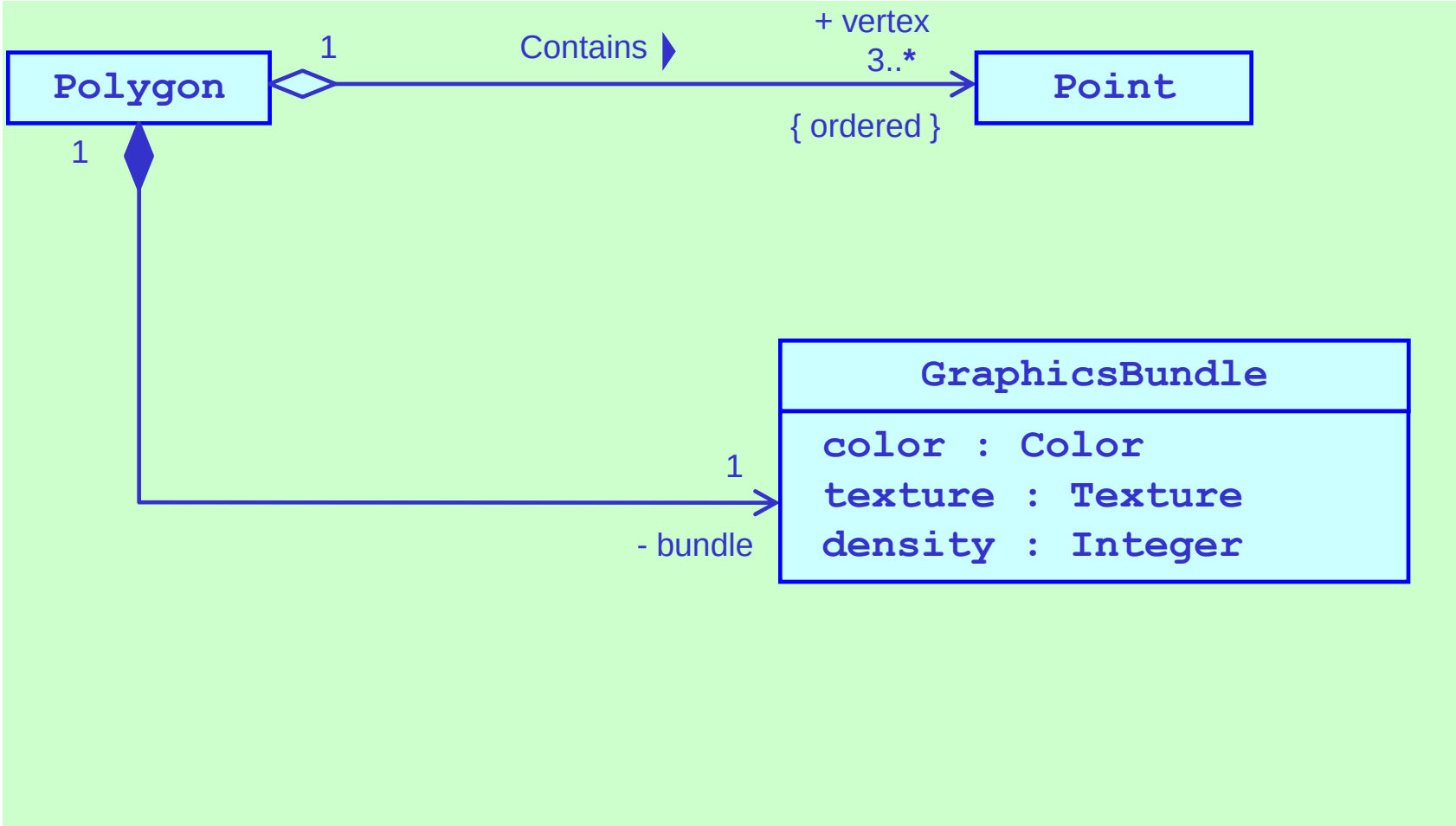

```
// Java
B x = new D();
T[] y = x.getA();
// y = b + d
```

```
// Java
B x = new B();
T[] y = x.getA();
// y = b
```

```
// Java
A x = new A();
T[] y = x.getA();
// y = {}
```

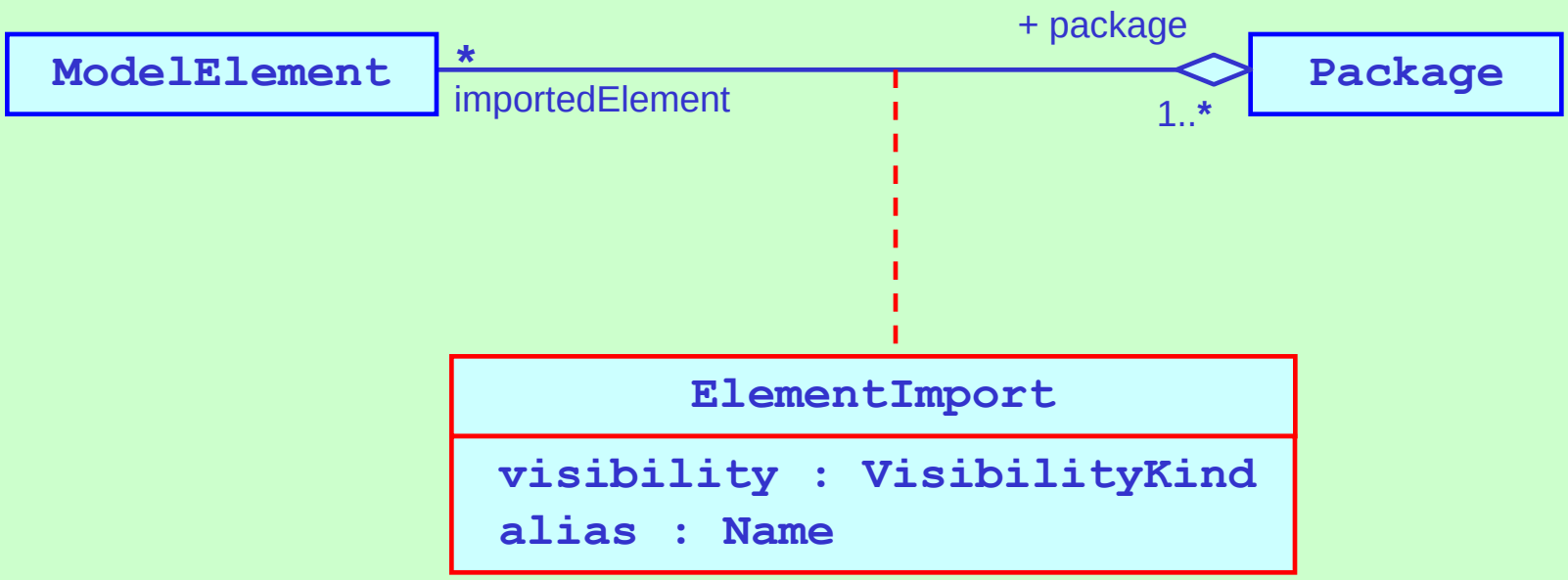
Отношение ассоциации.

Пример свойств окончания ассоциации



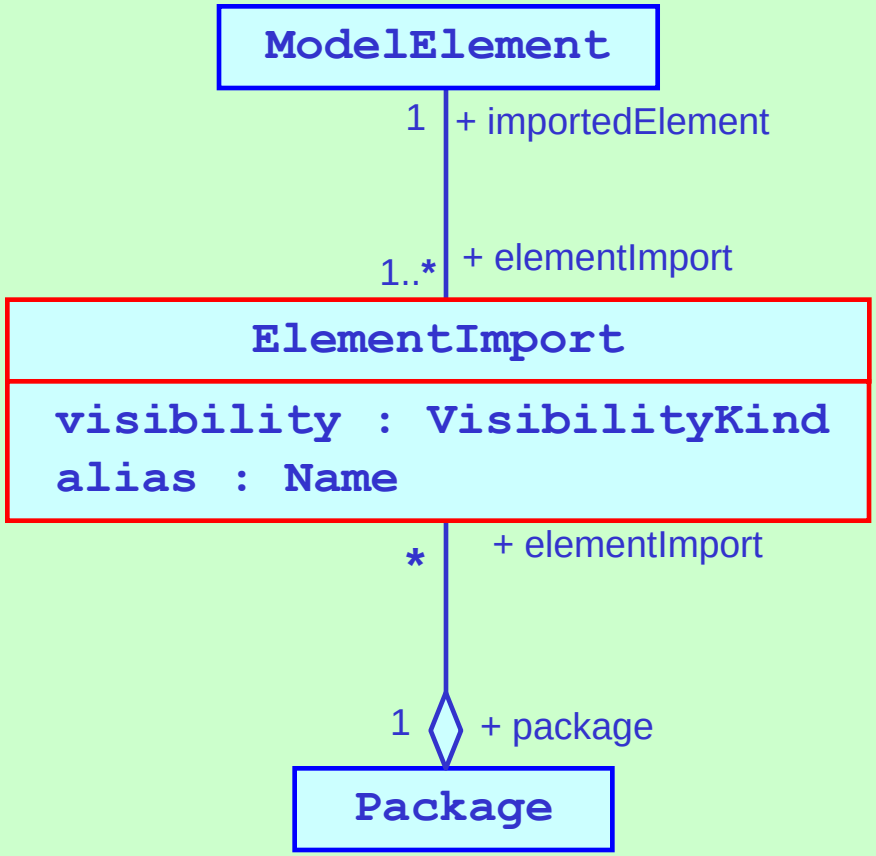
Отношение ассоциации.

Ассоциация - класс



Отношение ассоциации.

Упрощение ассоциации - класса



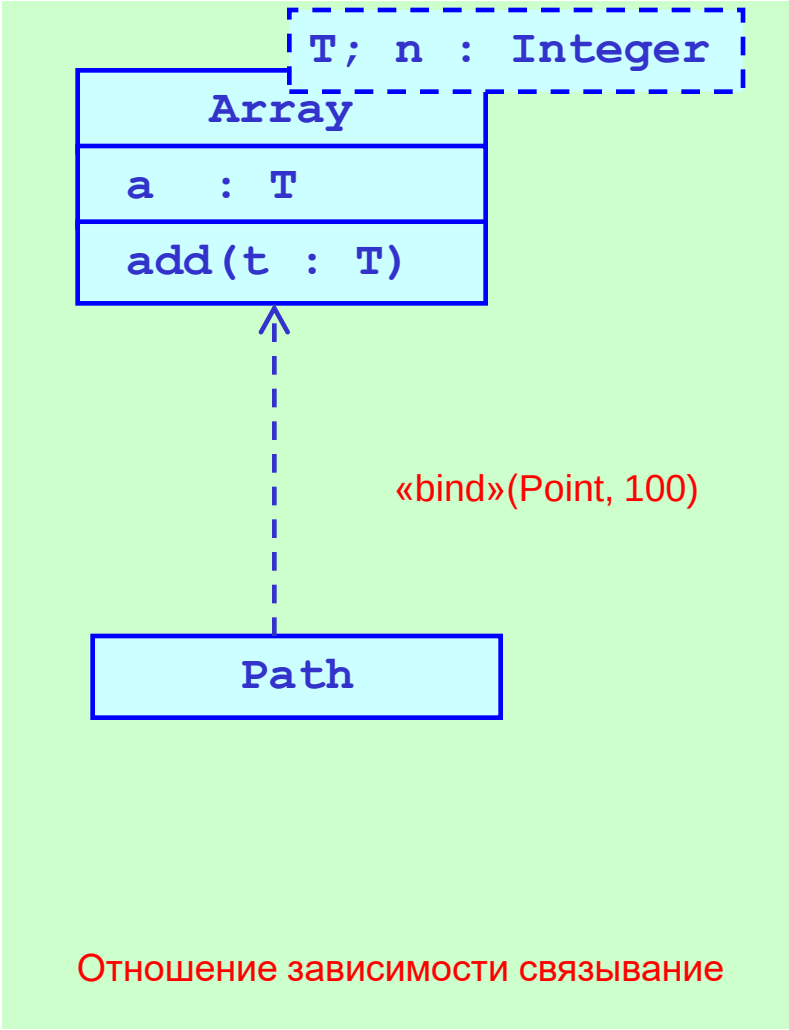
2.4 Отношение зависимости

- Отношения обобщения
- Отношения реализации
- Отношение ассоциации
- Отношение зависимости

Отношение зависимости. Категория зависимости “связывание”

- Категория зависимости связывание
- Категория зависимости абстракция
- Категория зависимости использование
- Категория зависимости разрешение

Отношение зависимости.
Изображение зависимости связывание



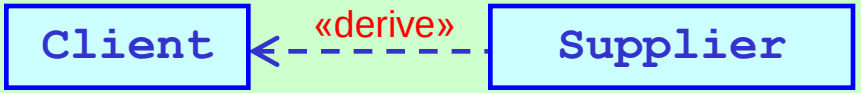

```
// C++
template <class T, int n>
class Array {
public:
   T a;
   void add(T t);
};
```

```
// C++
Array<Point, 100> path;
```

Отношение зависимости. Категория зависимости “абстракция”

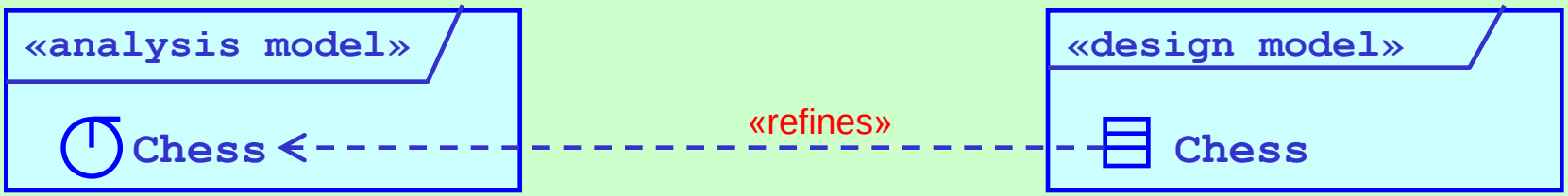
- Категория зависимости связывание
- Категория зависимости абстракция
- Категория зависимости использование
- Категория зависимости разрешение

Отношение зависимости.
Изображение зависимостей абстракции

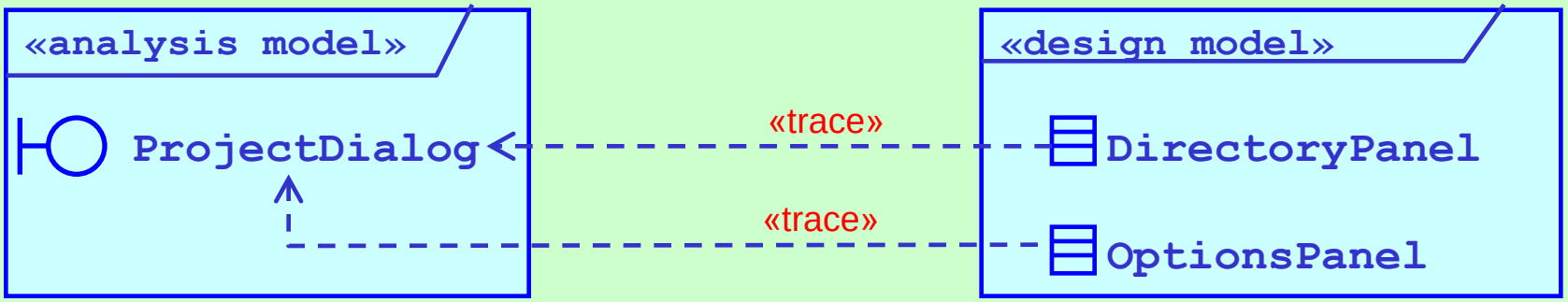


Порождение (derive) элемента модели-клиента из элемента модели-поставщика

Класс Chess из зависящей от языка реализации модели проектирования уточняет (refines) класс Chess из независящей от языка реализации модели анализа



Классы DirectoryPanel и OptionsPanel были созданы из модели проектирования были созданы из спецификации класса ProjectDialog модели анализа

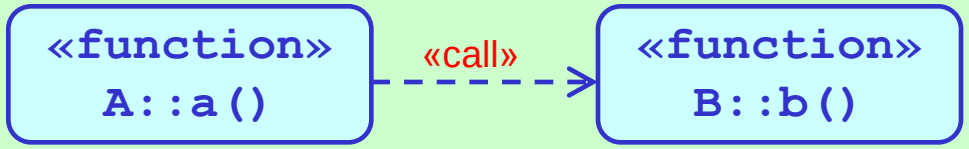


Отношение зависимости. Категория зависимости “использование”

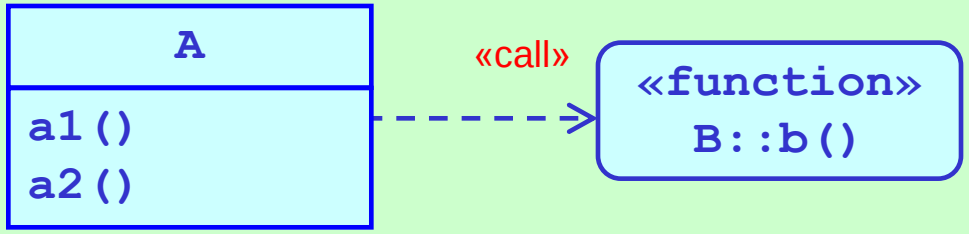
- Категория зависимости связывание
- Категория зависимости абстракция
- Категория зависимости использование
- Категория зависимости разрешение

Отношение зависимости.

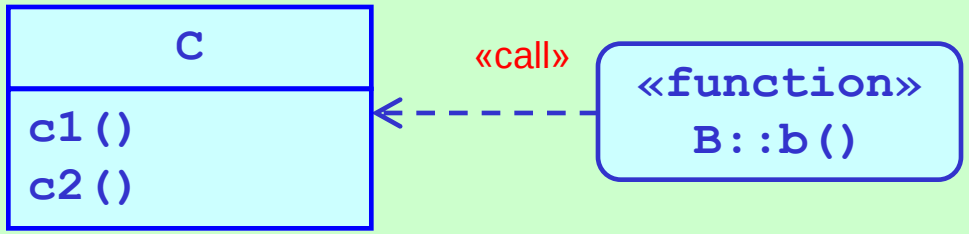
Изображение зависимости “вызов функции”



Представление графа вызова с помощью отношения зависимости. Порядок вызова не важен.



Представление всех функций классов вызывающих функцию B::g().

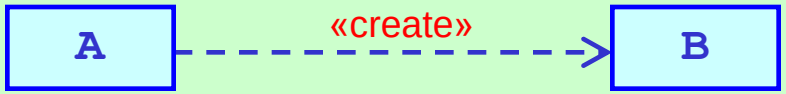


Представление всех функций классов вызываемых функцией B::g().

Изображения зависимости вызова могут быть полезны, например, как комментарии поясняющие вычисленные объектно-ориентированные метрики для функций программы.

Отношение зависимости.

Изображение зависимости “создание экземпляра”

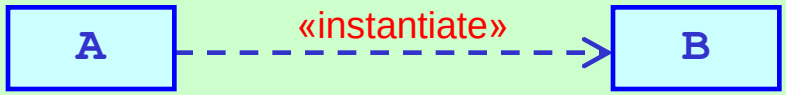


Создание экземпляра класса A приводит к созданию экземпляра класса B

```
// C#
public class A : B {
 public A(string s) : base(s) {}
}
```

```
// Java
public class A extends B {
   public A(string s) { base(s); }
}
```

```
// C++
class A : public B {
public:
  A(std::string s) : B(s) {}
}
```



Вызов методов класса A приводит к созданию экземпляра класса B

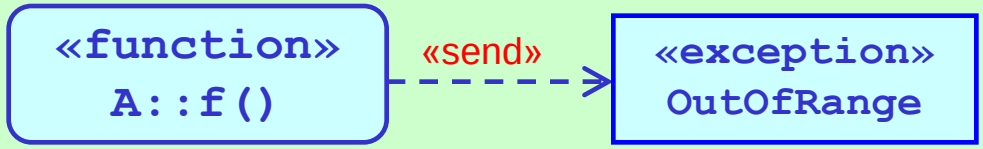
```
// C#
public class A : B {
 public B f(string s)
    { return new B(s); }
}
```

```
// Java
public class A extends B {
 public B f(string s)
    { return new B(s); }
}
```

```
// C++
class A : public B {
public:
  B f(std::string s)
    { return new B(s); }
}
```

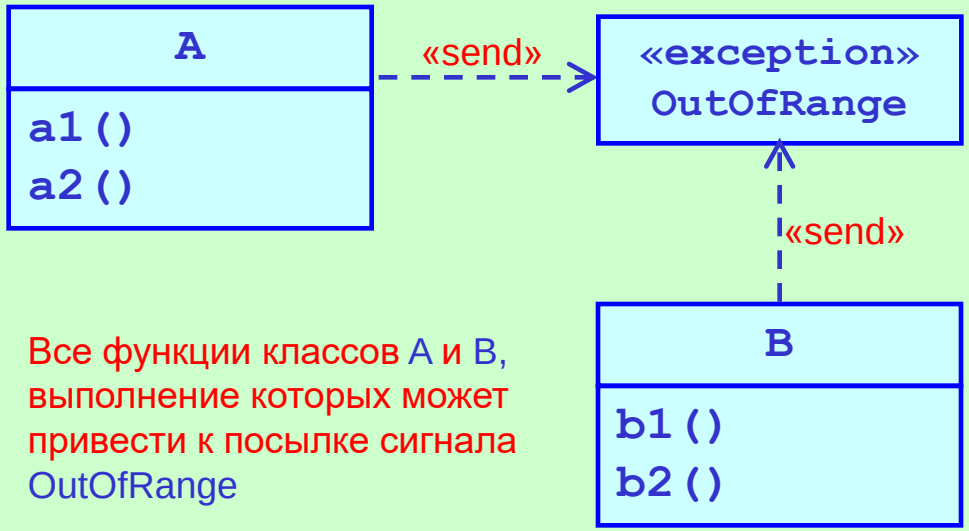
Отношение зависимости.

Изображение зависимости “посылка сигнала”



Выполнение функции A::f() может привести посылке сигнала OutOfRange

```
// Java
class OutOfRange
extends Exception
   {}

class A {
   void f()throws OutOfRange
       {}
}
```



Все функции классов A и B, выполнение которых может привести к посылке сигнала OutOfRange

```
// Java
class OutOfRange
extends Exception
   {}
class A {
 void a1()throws OutOfRange {}
 void a2()throws OutOfRange {}
}
class B {
 void b1()throws OutOfRange {}
 void b2()throws OutOfRange {}
}
```

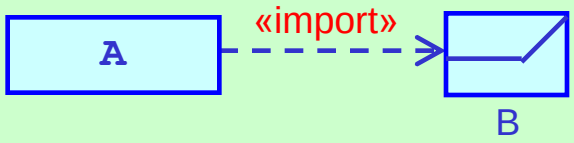
Отношение зависимости. Категория зависимости “разрешение”

- Категория зависимости связывание
- Категория зависимости абстракция
- Категория зависимости использование
- Категория зависимости разрешение

Отношение зависимости.
Изображение зависимости “импорт”

МГУ им. М.В.Ломоносова. Факультет ВМК. Романов Владимир Юрьевич ©2024

Отношение между пространствами имен. Содержимое пространства имен поставщика полностью или частично копируется в пространство имен клиента.



Импорт всего пространства имен **B**.

```
//Java
import B.*;

public class A {}
```

```
// C++
#include “B.h”
using namespace B;

class A {};
```

```
// C#
using B;


public class A {}
```

Импорт класса **C** из пространства имен **B**.

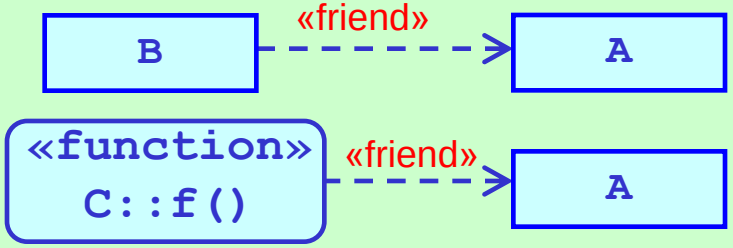
```
//Java
import B.C;

public class A {}
```

```
// C++
#include “B.h”
using B::C;

class A {};
```

```
// В C# импорт класса возможен
// только с новым именем-псевдонимом.
using C_from_B = B.C;

public class A {}
```

Отношение зависимости.
Изображение зависимости “друг”

Клиенту отношения дается доступ ко всем атрибутами и методов класса-поставщика независимо от типа видимости атрибутов и методов.



Доступ всех функций класса **B** и функции **f** класса **C** ко всем атрибутам и функциям класса **A**.

```
// C++
class B, C;
class A {
   friend B;
   friend C.f;
   private:   int x;
   protected: int y;
};
```

```
class C {
   A a;

   void f() {
      a.x = 1; // Допустимо.
      a.y = 2; // Допустимо.
   }
   void g() {
      a.x = 1; // Не допустимо.
      a.y = 2; // Не допустимо.
   }
};
```

```
class B {
   A a;

   void f() {
      a.x = 1; // Допустимо.
      a.y = 2; // Допустимо.
   }
   void g() {
      a.x = 1; // Допустимо.
      a.y = 2; // Допустимо.
   }
};
```