IT Systems Implementation –

UML Class diagrams II,

and implementing classes and their structural relations in Java

Bettina Berendt

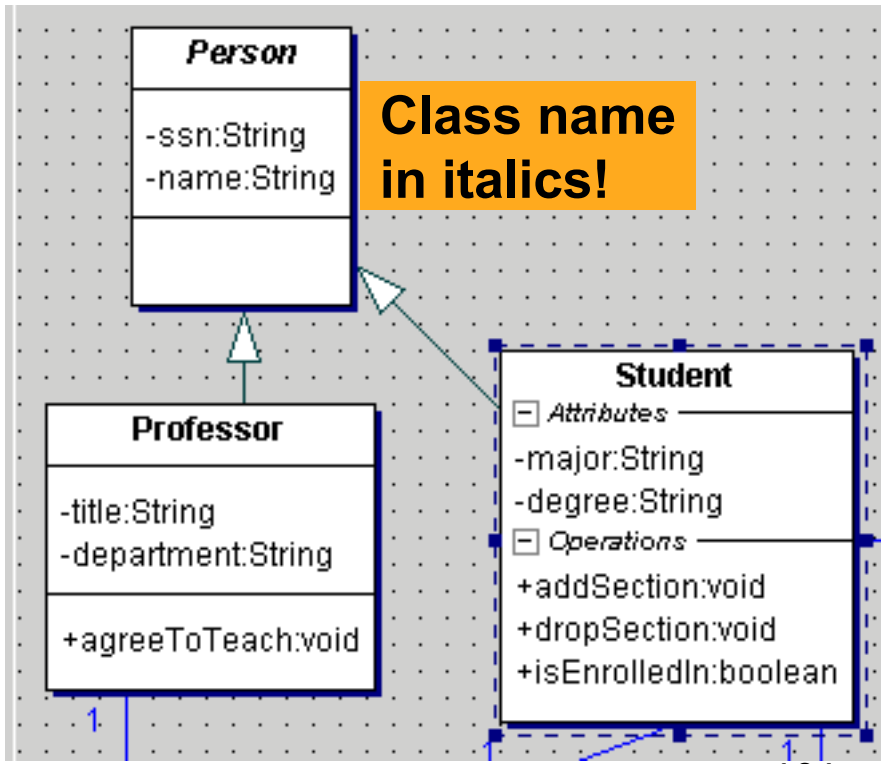
Humboldt-Universität zu Berlin, Institut für Wirtschaftsinformatik
http://www.wiwi.hu-berlin.de/~berendt/lehre/2002w/isi/
(Based on Barker, Chs. 10 and 14, and Together 6.0.1)

Agenda

- Additional refinements of UML class diagrams
- UML ➔ Java conversion
- The *Together* software

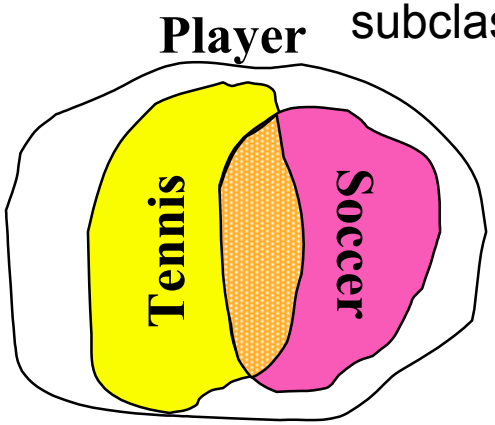
Abstract classes

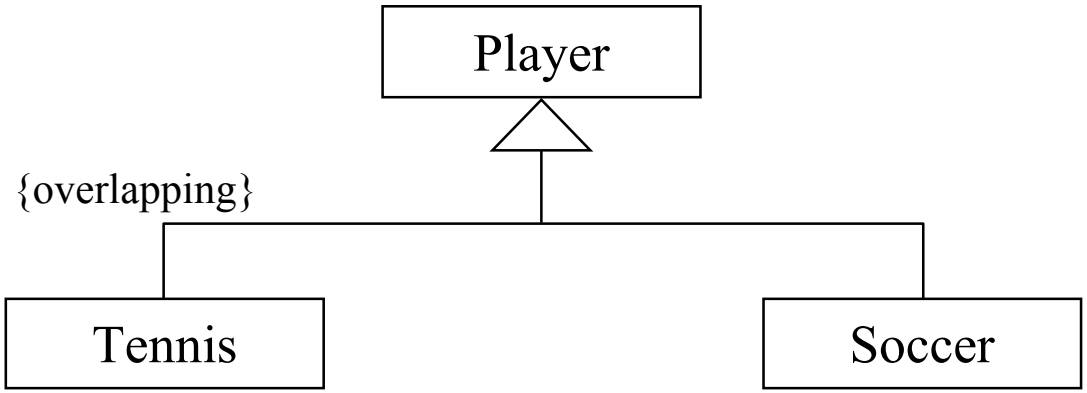
- No-one is „just a person“. Everyone is either a student, or a professor, or ...
- An abstract class is one that cannot be instantiated.
- It only serves to define all attributes and behaviours that all subclasses (or their instances) have in common.



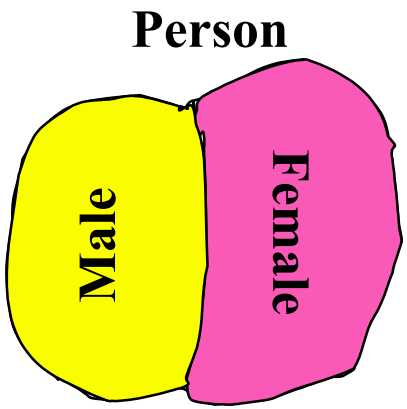
GENERALIZATION — COVERAGE

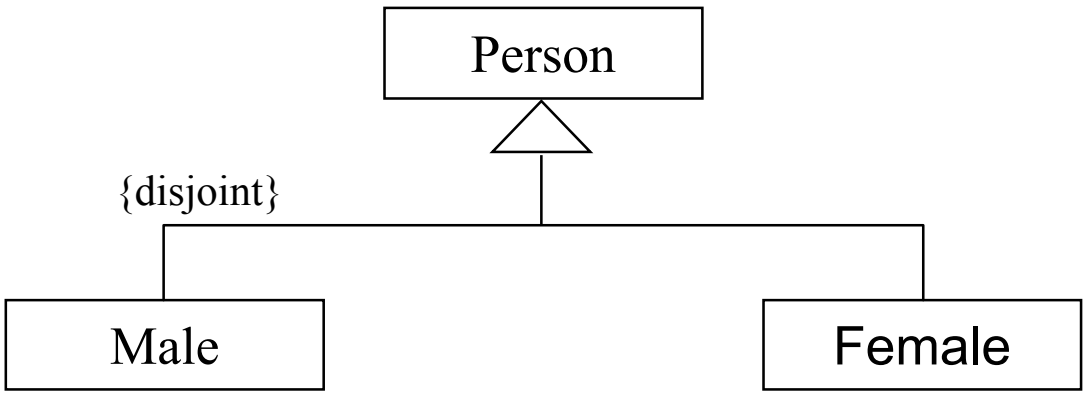
overlapping - a superclass object can be a member of more than one subclass





disjoint - a superclass object is a member of at most one subclass

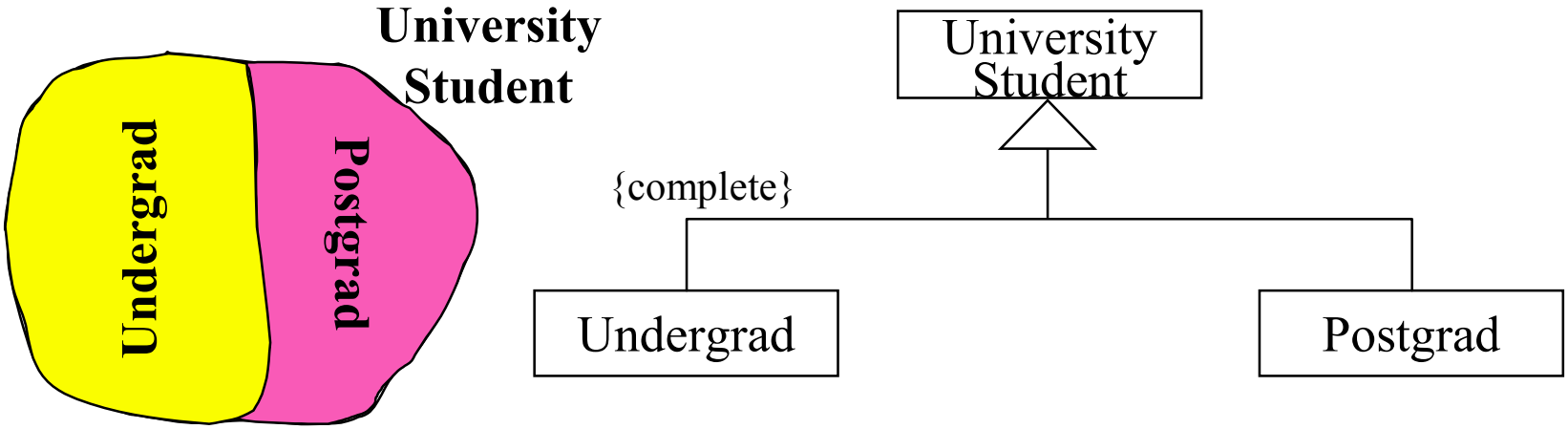




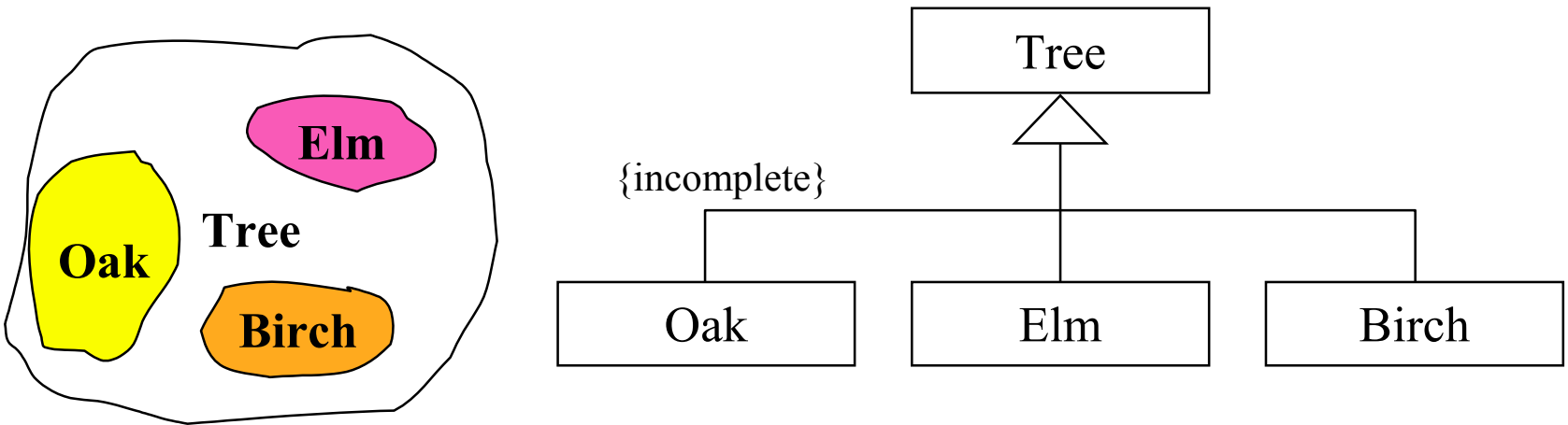
(from http://course.cs.ust.hk/comp211/2002Spring/ Slides/02OOModeling.ppt)

GENERALIZATION — COVERAGE (cont'd)

complete - all superclass objects are also members of some subclass



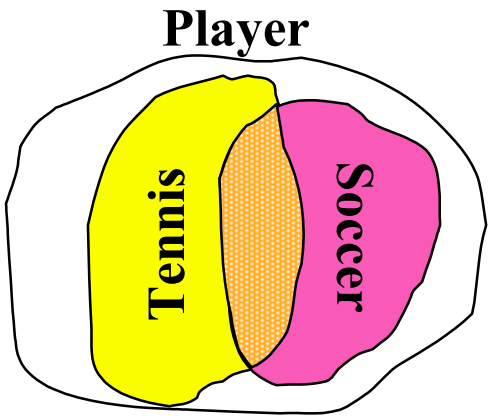
incomplete - some superclass object is not a member of any subclass

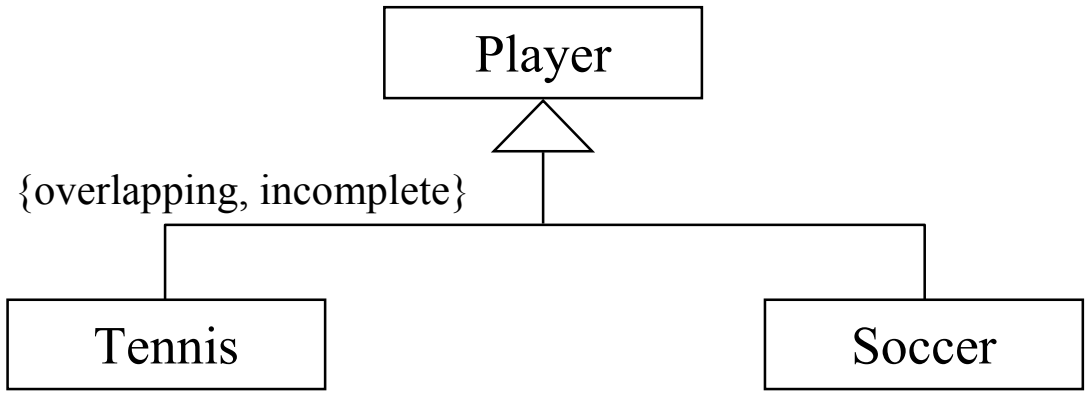


(from http://course.cs.ust.hk/comp211/2002Spring/ Slides/02OOModeling.ppt)

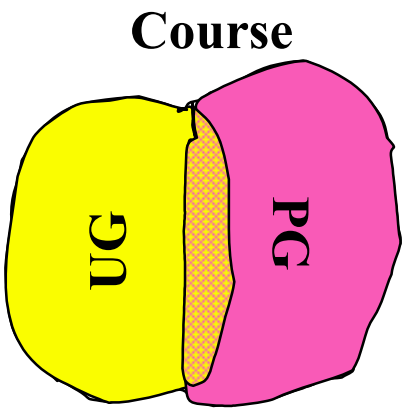
GENERALIZATION — COVERAGE (cont'd)

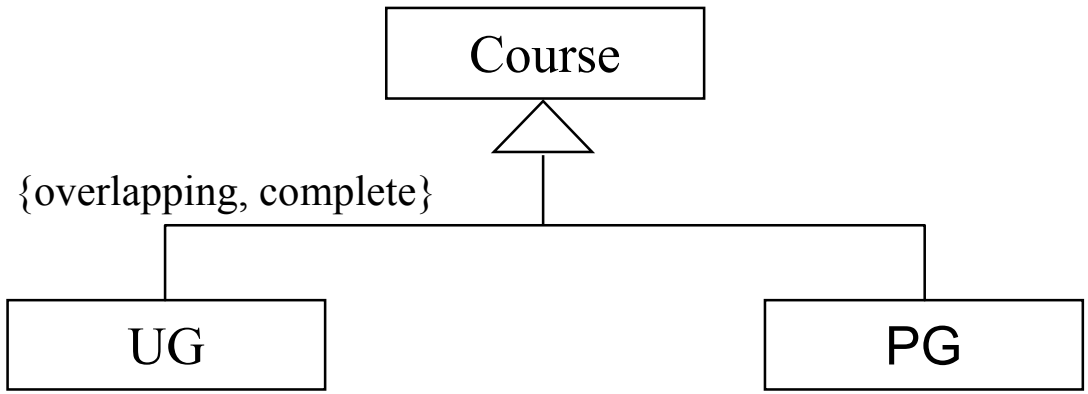
overlapping, incomplete





overlapping, complete

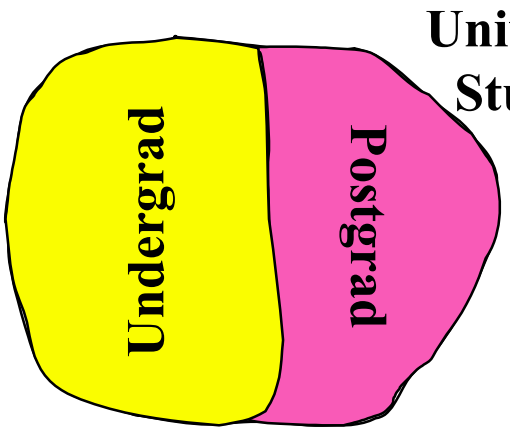


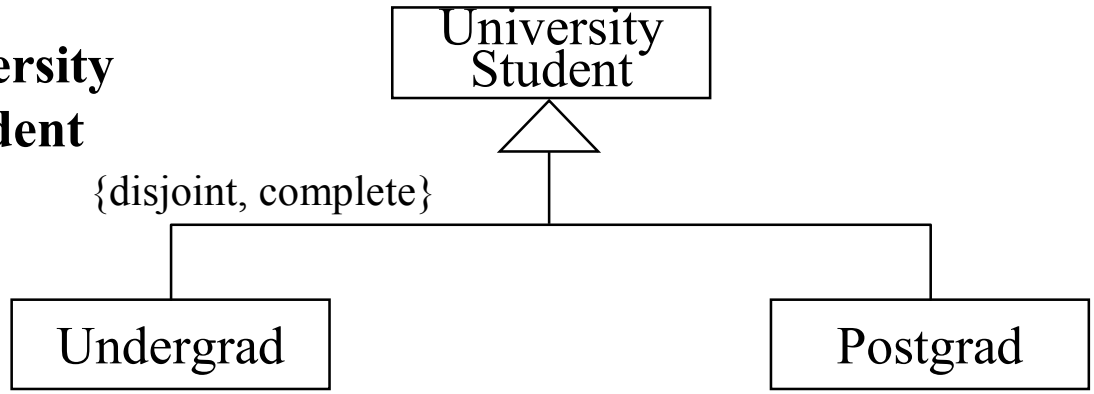


(from http://course.cs.ust.hk/comp211/2002Spring/ Slides/02OOModeling.ppt)

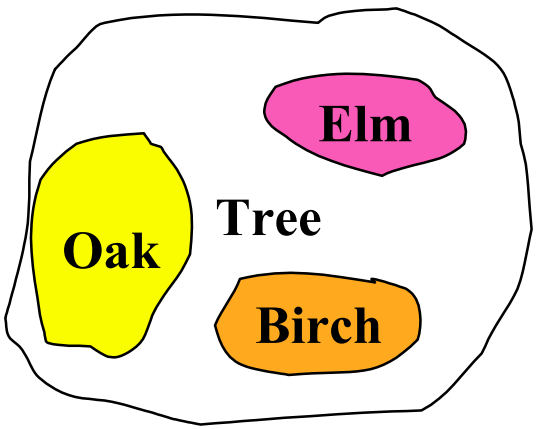
GENERALIZATION — COVERAGE (cont'd)

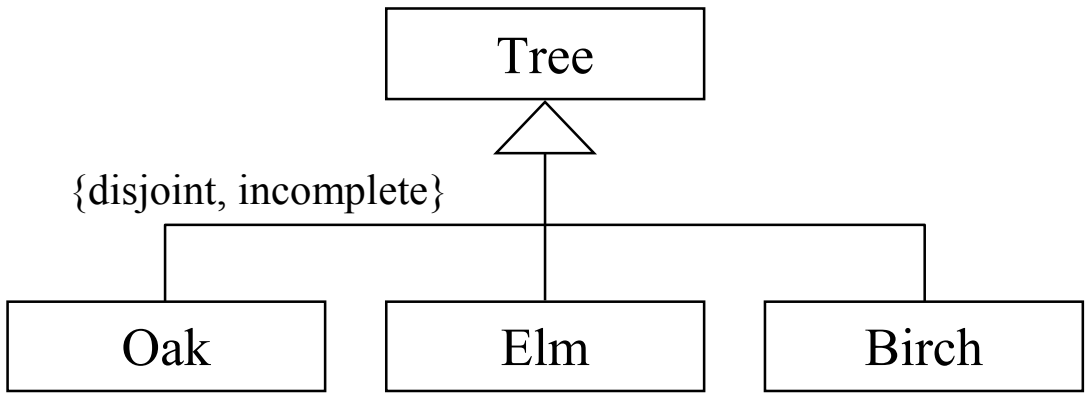
disjoint, complete





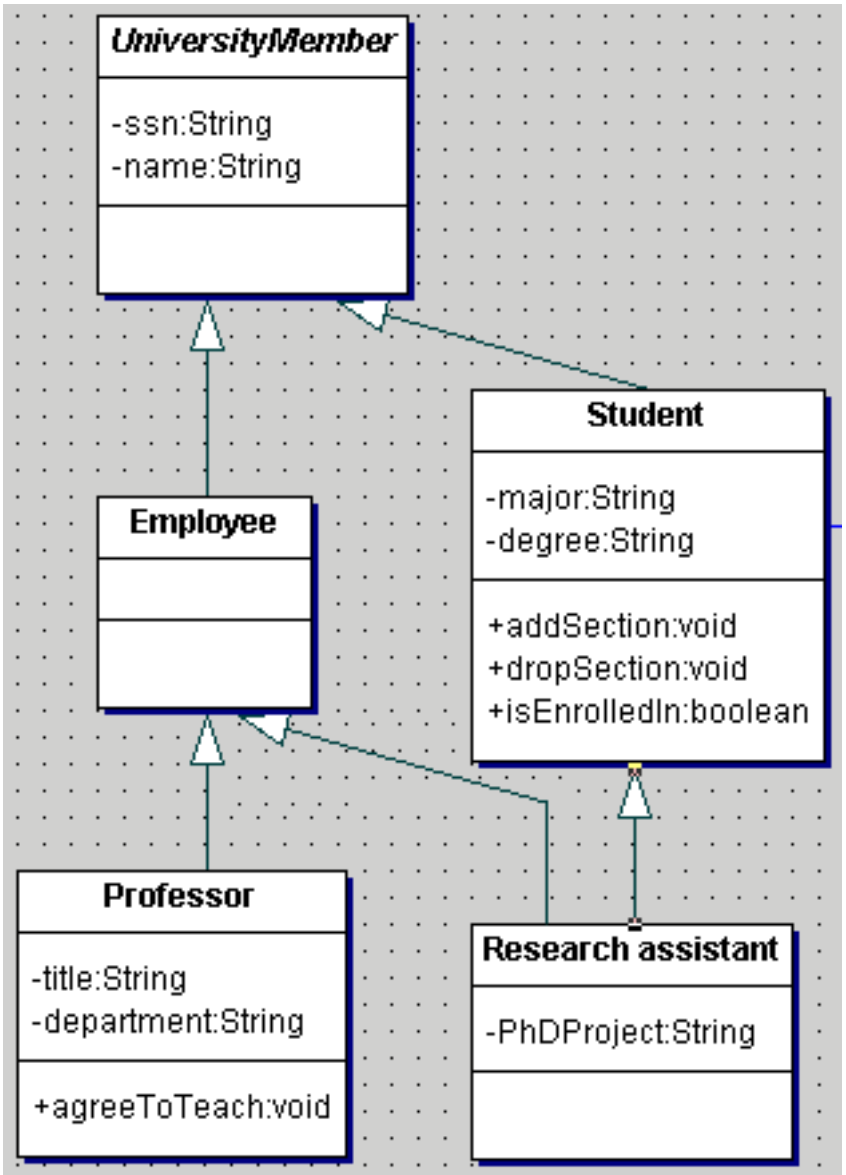
disjoint, incomplete





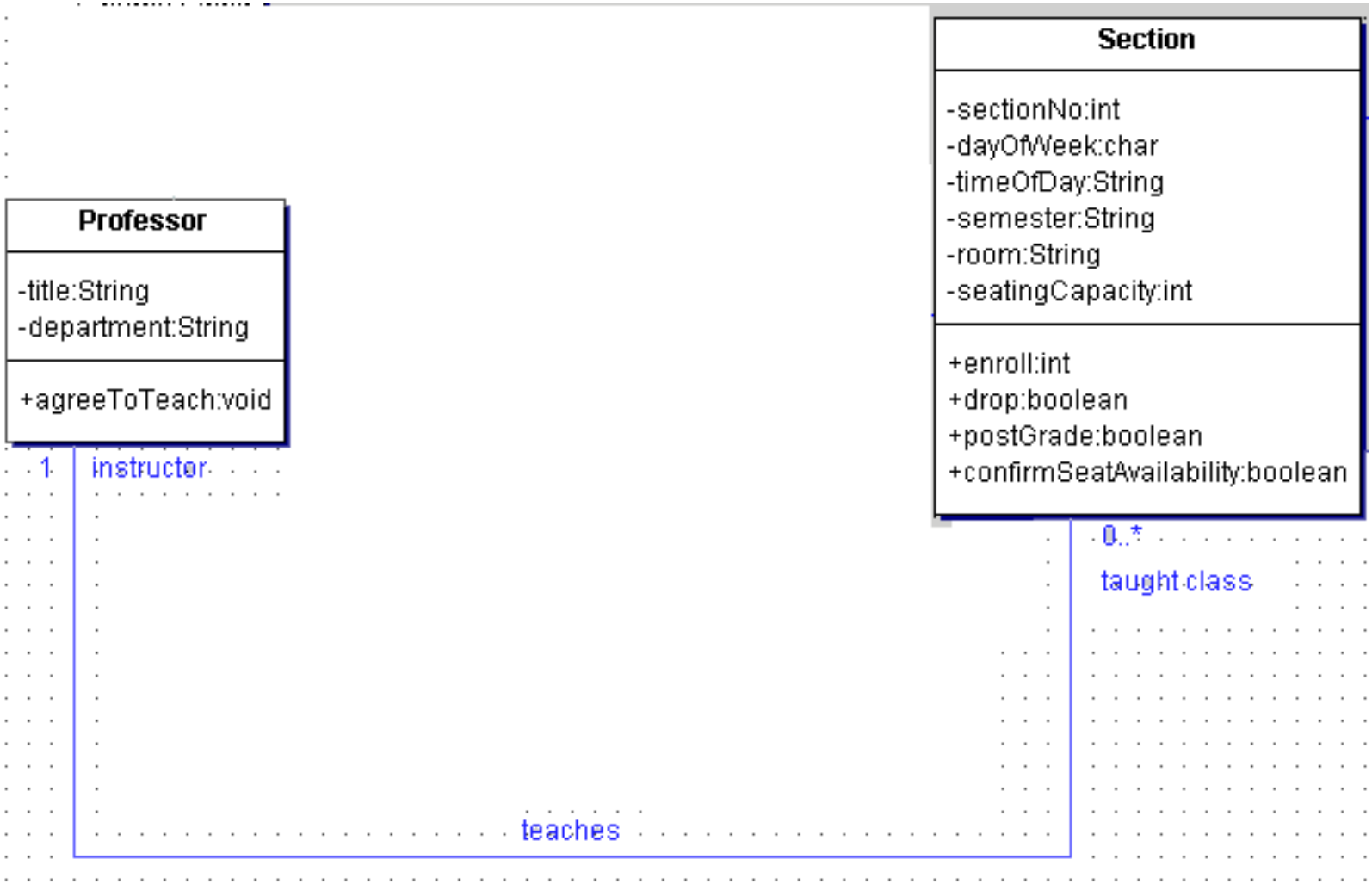
(from http://course.cs.ust.hk/comp211/2002Spring/ Slides/02OOModeling.ppt)

Multiple inheritance



Not supported in Java!

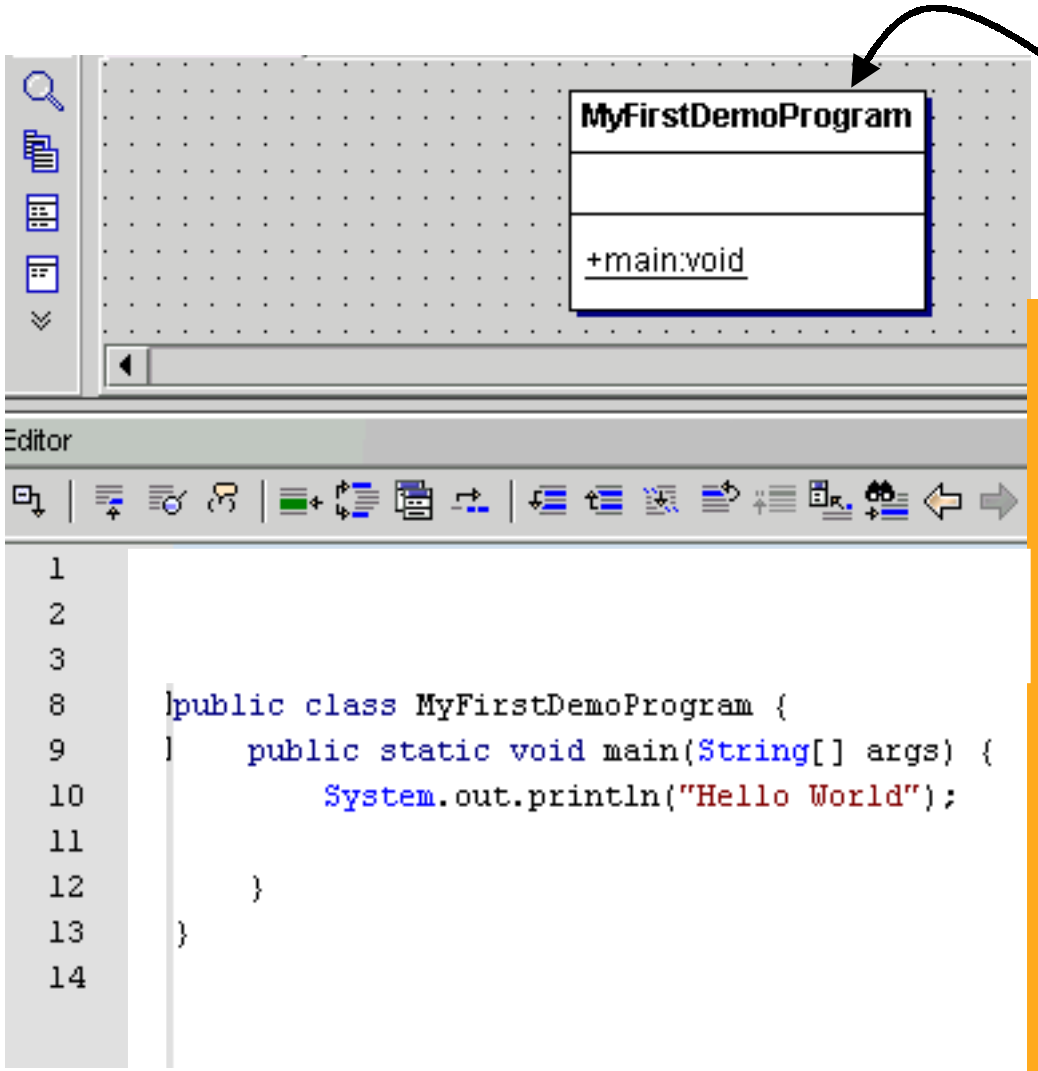
Roles in a structural relation



Agenda

- Additional refinements of UML class diagrams
- UML ➔ Java conversion
- The *Together* software

A program with just one solution space class (the one holding the `main` method)

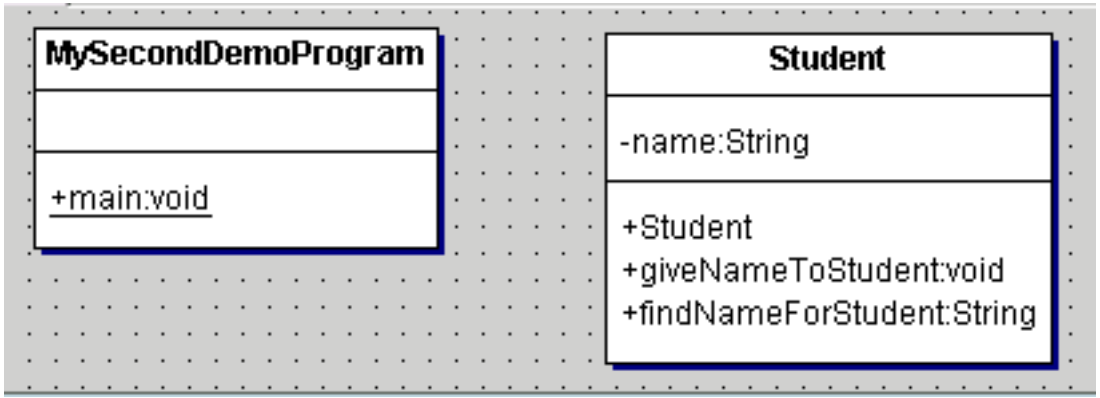


The program's class structure as a UML class diagram

The simplest Java program

- 1 file: `MyFirstDemoProgram.java`
- which contains 1 class: `MyFirstDemoProgram`
- which contains 1 method: `main`
- which contains 1 command: print „`Hello World`“ to the screen

A program with one solution space class and one domain space class



Main purpose

of this program:

- **Show simple methods for a user-defined class (`Student` with attribute `name`, which can be set or retrieved)**
- **Show simple interaction between two classes**
- **Show differences in the Java treatment between**
  - **Simple datatypes (integer numbers `int` as an example; these are not classes in Java)**
  - **The datatype `String`, which is a standard SDK class (i.e., some simplifications are provided for handling this datatype)**
  - **A typical user-defined datatype (a class)**

Demo program #2: Java source code of `Student`

Other classes can't access the attribute `name` directly ...

... they have to use the publicly available functions for this.

```
public class Student {
    private String name; // the attribute

    public void giveNameToStudent(String n) {
        name = n;
    }
    public String findNameForStudent() {
        return name;
    }

     public Student() { // a constructor
        // ...
    }
}
```

Every class can request the construction of a `student` object

Demo program #2: Java source code of the “driver” class (1)

```
public class MySecondDemoProgram {
  public static void main (String[] args) {
    System.out.println("First message");

    int i;
    i = 4;
    System.out.println("The variable i is first: " + i);
    i = i + 3;
    System.out.println("The variable i is now: " + i);

    String x;
    x = "One word";
    System.out.println("The variable x is first: " + x);
    x = x + " another word";
    System.out.println("The variable x is now: " + x);
```

Demo program #2: Java source code of the “driver” class (2)

(repeated from previous slide)

```
        String x;
        x = "One word";
        System.out.println("The variable x is first: " + x);
        x = x + " another word";
        System.out.println("The variable x is now: " + x);

        Student s = new Student();
        s.giveNameToStudent("Alberta");
        String nn = new String();
        nn = s.findNameForStudent();
        System.out.println("The student is called " + nn);
    }
}
```

Demo program #2: output

```
First message
The variable i is first: 4
The variable i is now: 7
The variable x is first: One word
The variable x is now: One word another word
The student is called Alberta
```

Compiling and running a Java application: From the command line

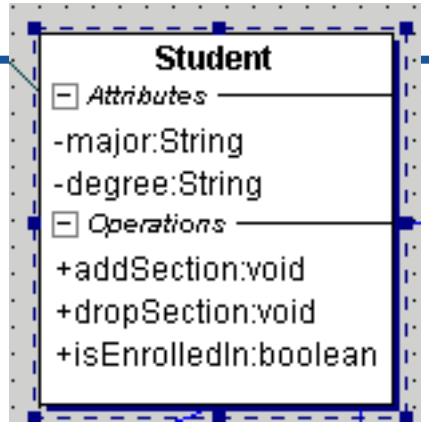
- From the command line (MS DOS window or Unix shell): compile the program

```
% javac MyProgram.java
```

This generates the file MyProgram.class. Then run it:

```
% java MyProgram [optional command line arguments]
```

UML ➔ Java: Classes with attributes and operations (called “methods” in Java)

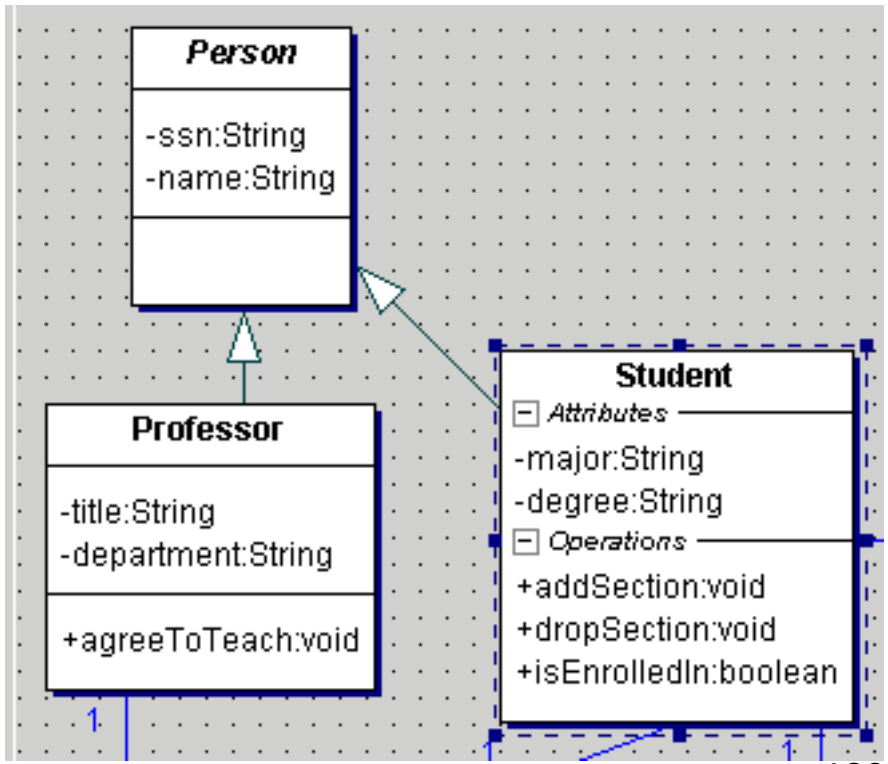



```
public class Student {
    private String major;
    private String degree;

    public void addSection(Section s) {
    }

    public void dropSection(Section s) {
    }

    public boolean isEnrolledIn(Section s) {
    }
}
```

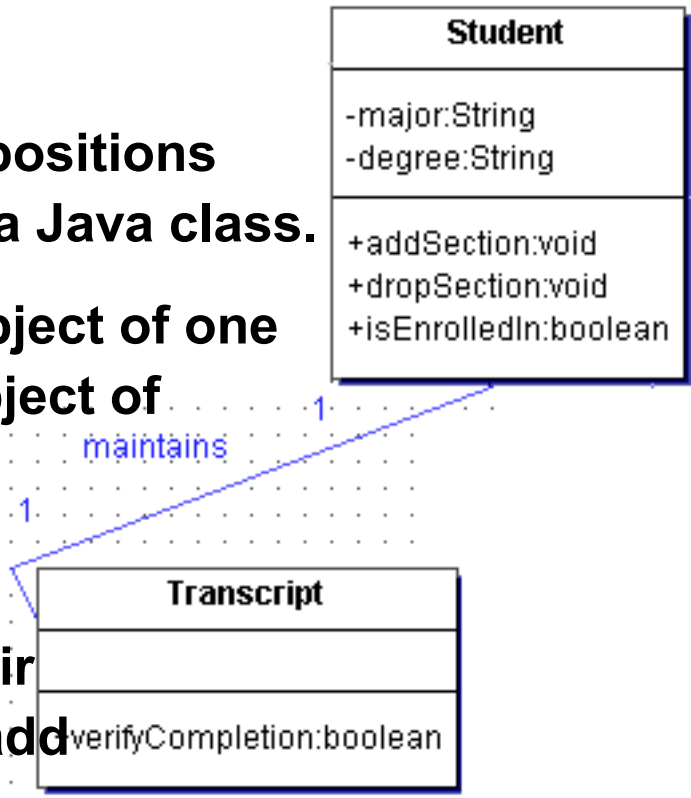
Attributes can be put before or after methods; most programmers put them first.

UML ➔ Java: Inheritance

```
public class Student extends Person {
    // ...
}
public class Professor extends Person {
    // ...
}
```



UML → Java: Structural relations – one-to-one



- **Associations, aggregations, and compositions must be implemented as attributes in a Java class.**
- **An attribute is needed whenever an object of one class needs to access its „partner“ object of the other class.**

Example: the `maintains` association

- **If `Student` objects need to access their corresponding `Transcript` objects, add**

```
private Transcript transcript;
```

to the `Student` class

- **If `Transcript` objects need to access their corresponding `Student` objects, add**

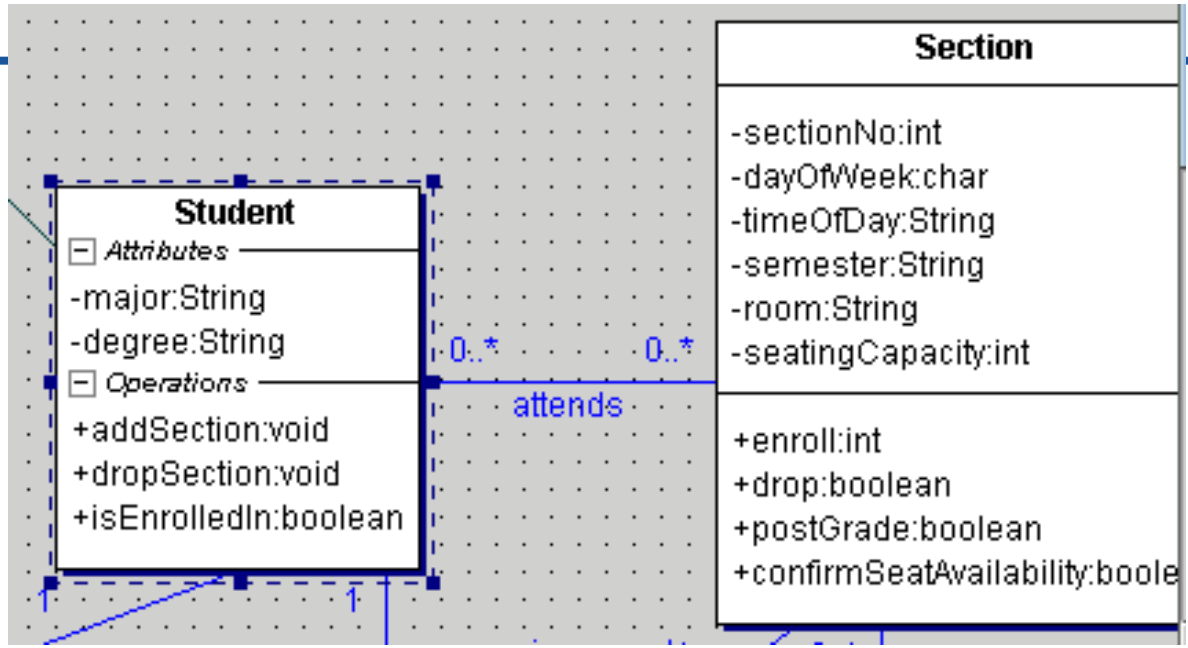
```
private Student studentOwner;
```

to the `Transcript` class

UML → Java: Structural relations – *x*-to-many (1)

- But what if the relation is one-to-many or many-to-many?
- Example: the `attends` association between `Student` and `Section`
- Solution: use a collection type, e.g.
  - a `Vector`
  - a `Hashtable`
- (both in the package `java.util`)

UML ➔ Java: Structural relations – *x*-to-many (2)



- If `Student` objects need to access their corresponding `Section` objects, we can add

  `private Vector attends;` to the `Student` class

- If `Section` objects need to access their corresponding `Student` objects, we can add this to the `Section` class:

  `private Hashtable enrolledStudents;`

UML → Java: Association classes and *n*-ary associations

- Java cannot represent association classes or n-ary associations directly
- So while they help to create a better and less cluttered UML diagram, they have to be broken up into binary associations for translation into Java!
- An example is the class TranscriptEntry, which turns from an association class of the association attends to a class which has one association with the Student class and one with the Section class.

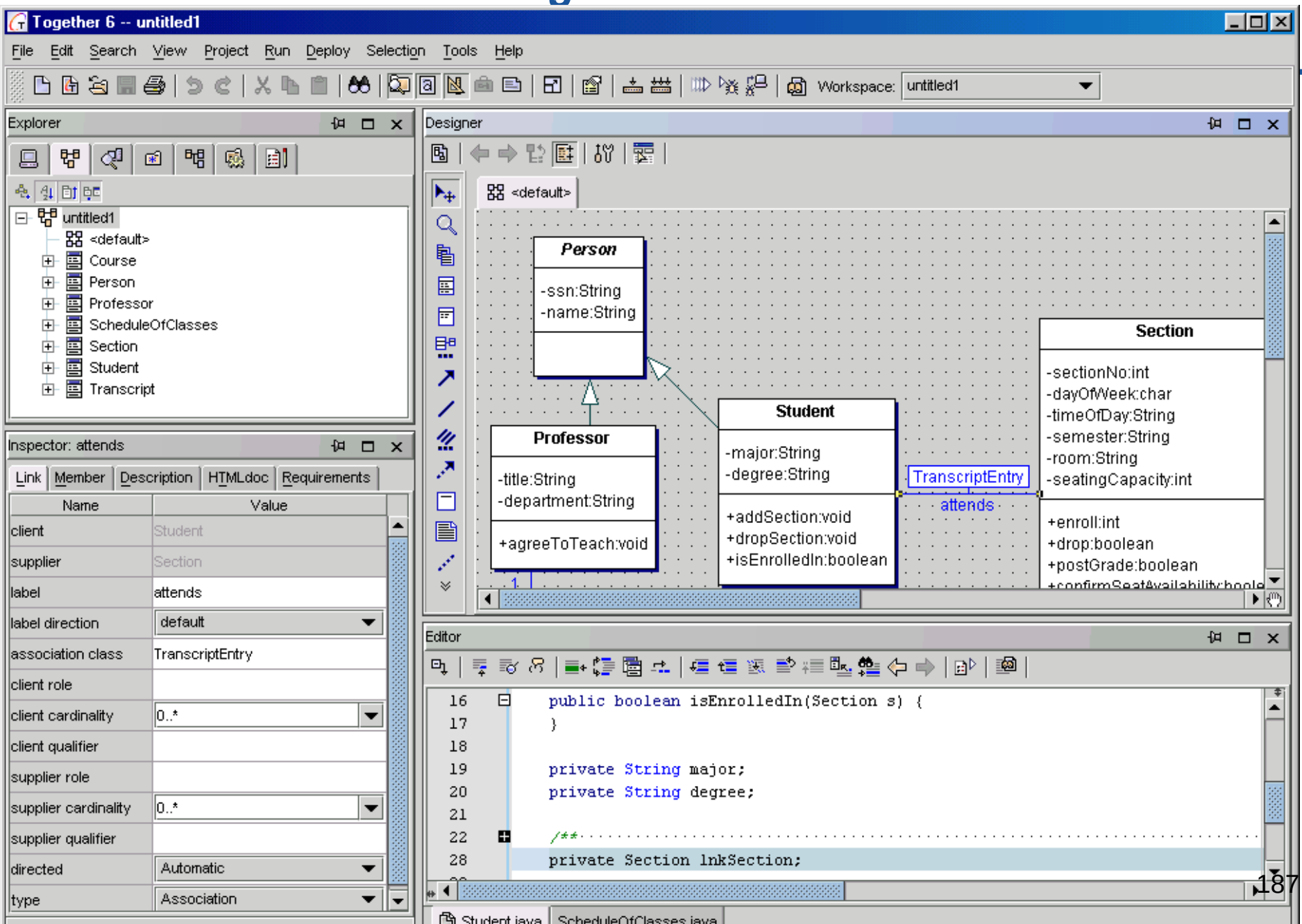
Agenda

- Additional refinements of UML class diagrams
- UML ➔ Java conversion
- The *Together* software

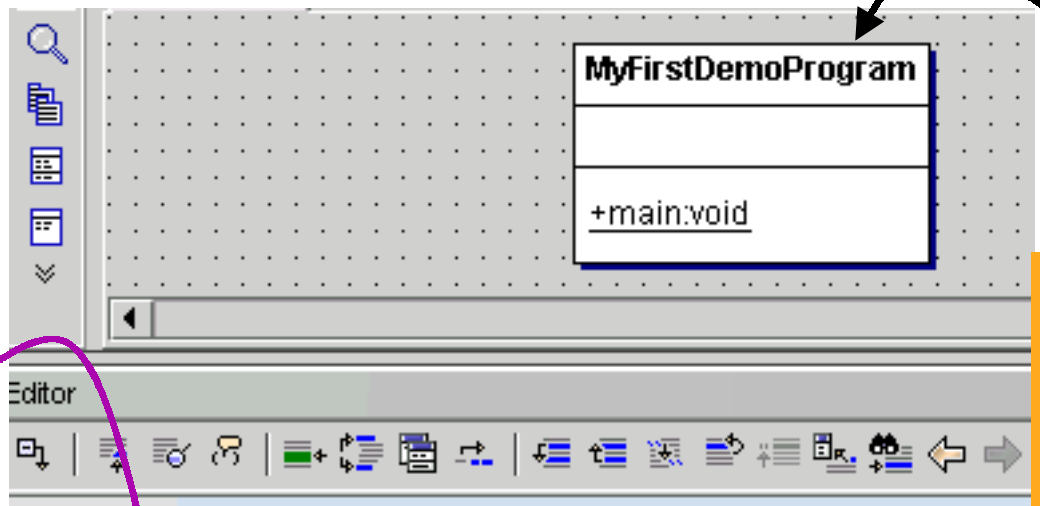
Overview (for more detail, see http://www.togethersoft.com)

- All UML diagrams can be drawn with any drawing program
- The correspondence UML diagram element – Java code element is not always unique and must be learned anyway.
- Still, tools can be helpful in supporting software and program design.
- Together is a tool that supports the design of UML diagrams
- It generates Java code from class diagrams and vice versa
- Like every tool, it has advantages and disadvantages
- The major disadvantages are
  - Handling is not 100% trivial
  - Some specifics are not 100% UML as found in textbooks (e.g., it differs from the UML used in the books used for this lecture)
  - Tool behaviour requires some reflection to be understood

A screenshot of the *Together* software in action



UML and Java in *Together*: a program with just one solution space class (the one holding the `main` method)



The same as a UML class diagram

Details hidden

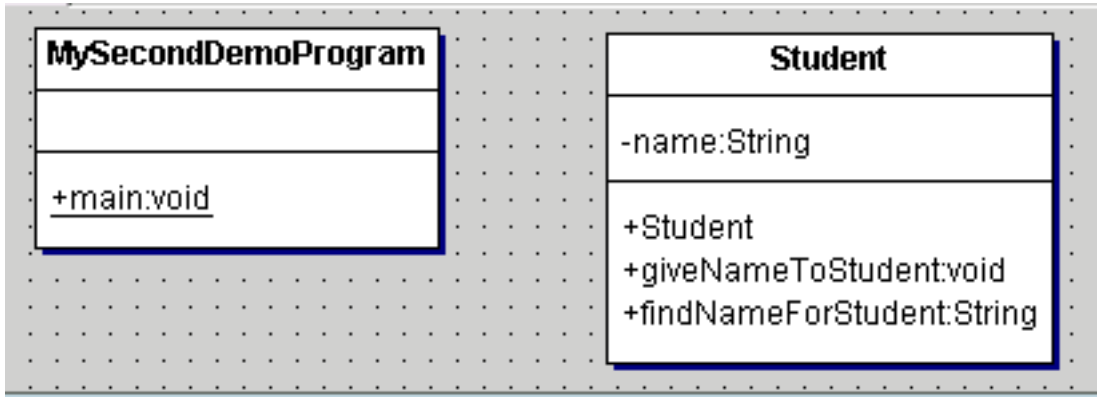
All detail shown

// Commented out: Print out the first command line argument

The simplest Java program

- **1 file:** `MyFirstDemoProgram.java`
- **which contains 1 class:** `MyFirstDemoProgram`
- **which contains 1 method:** `main`
- **which contains 1 command: print „`Hello World`“ to the screen**

UML and Java in *Together*: a program with one solution space class and one domain space class



Compiling and running a Java application: From within *Together*

- **F9 (or “Run”) compiles and runs the program.**

Demo program #2: output on the *Together* "command line"

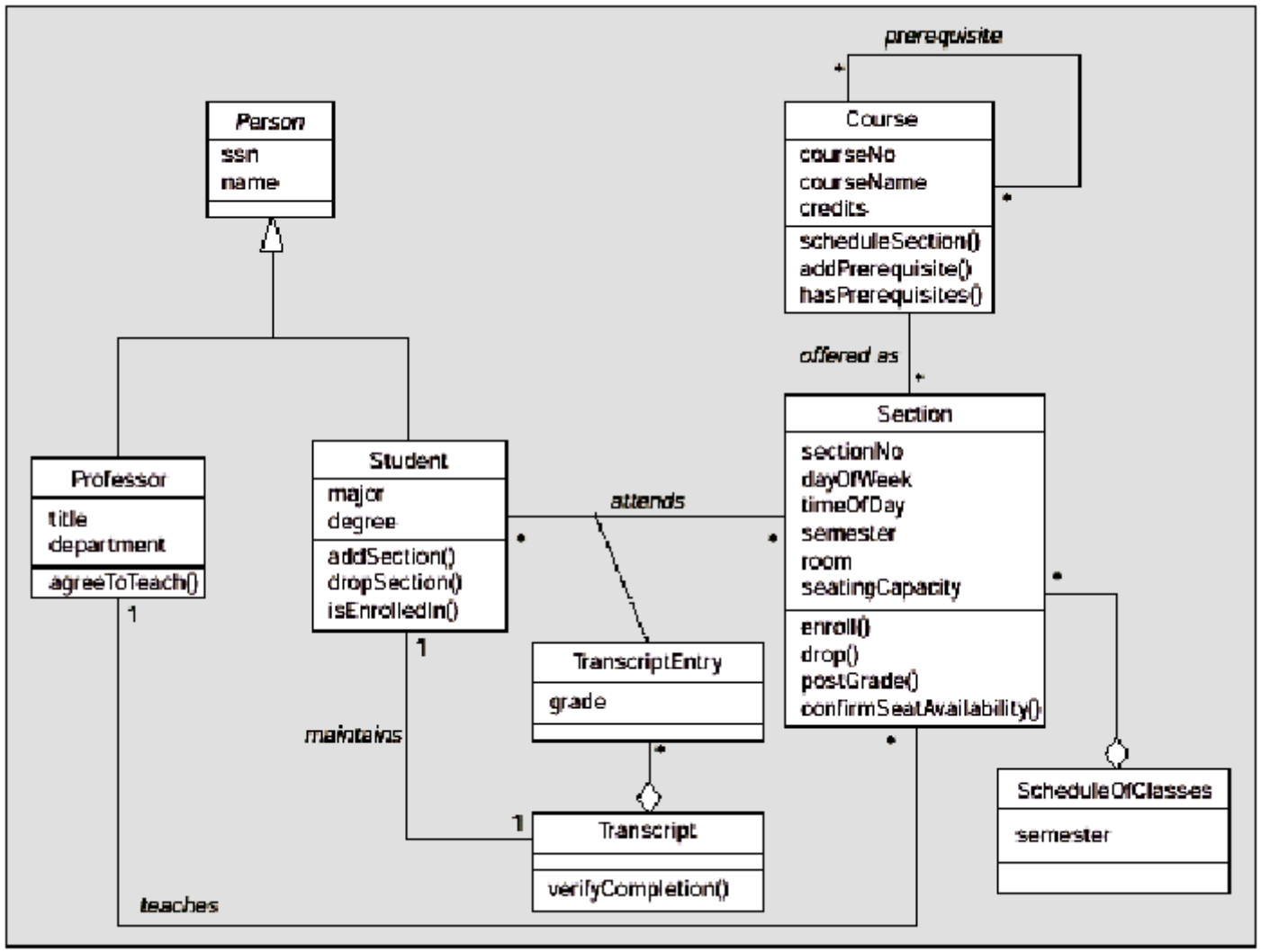
... (not shown): compilation ...

The JDK is the one that is „really" running the Java program

```
C:\jdk1.3.1_01\bin\javaw -classpath
C:\TogetherSoft\Together6.0.1\out\classes\MySecondD
emoProgram;C:\TogetherSoft\Together6.0.1\lib\javax.
jar; MySecondDemoProgram
```

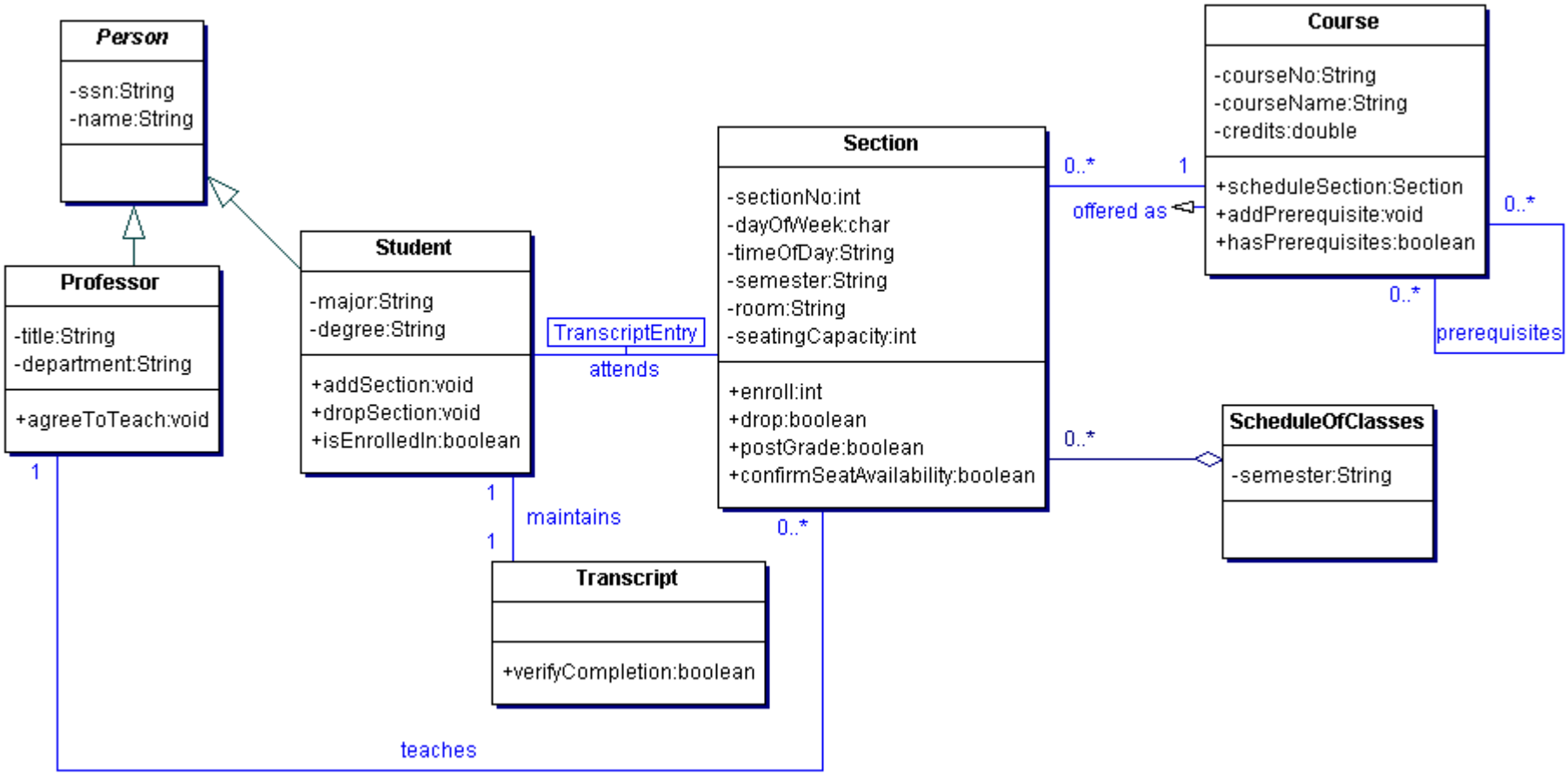
```
First message

The variable i is first: 4

The variable i is now: 7

The variable x is first: One word

The variable x is now: One word another word

The student is called Alberta
```

The UML class diagram of a Student Registration System (with attributes and operations)

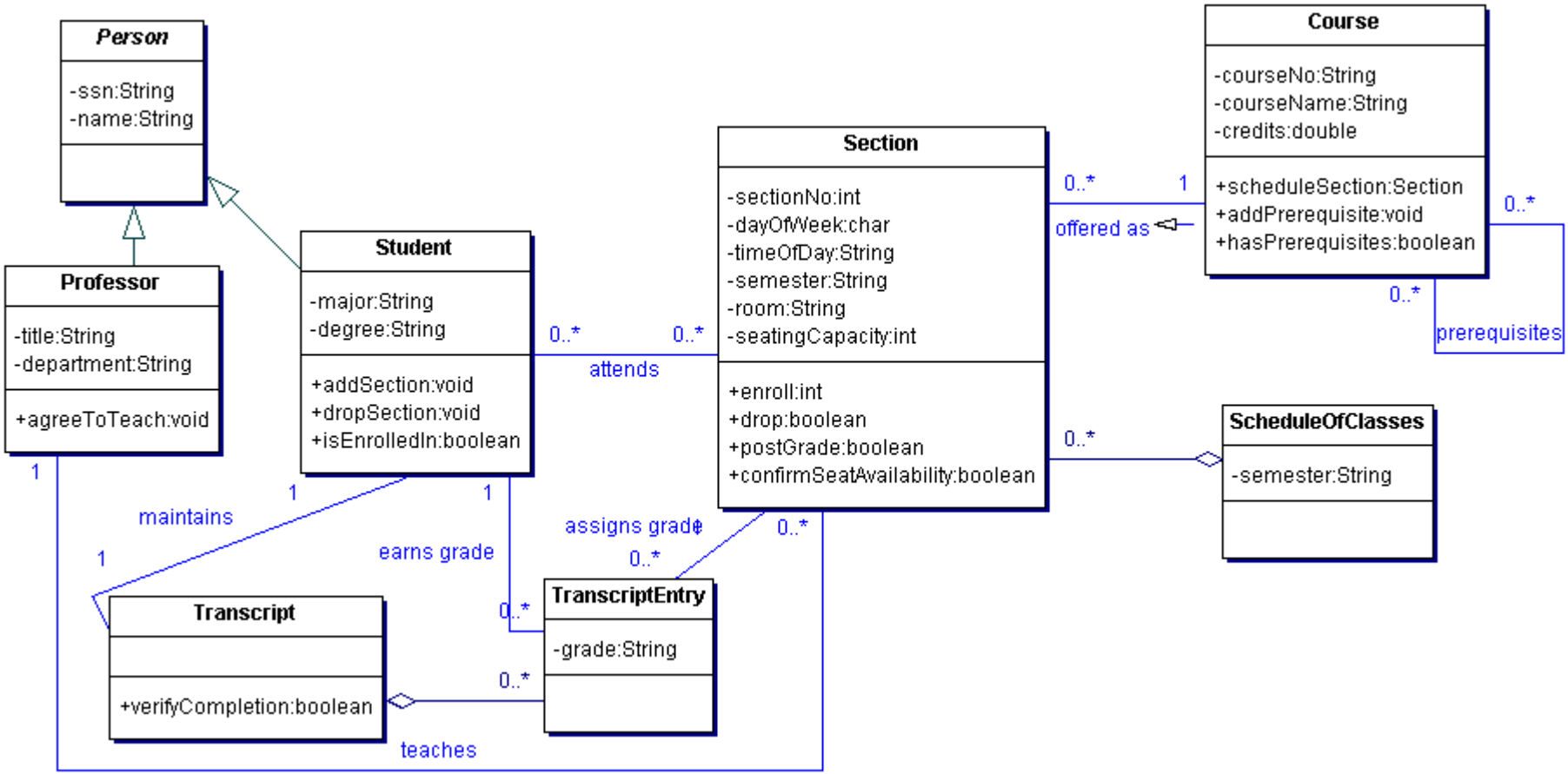


(adapted from Barker, p. 377)

The *Together* UML class diagram A: the one that corresponds most closely to the original SRS diagram



The *Together* UML class diagram B: The one that splits up an association class into two binary associations



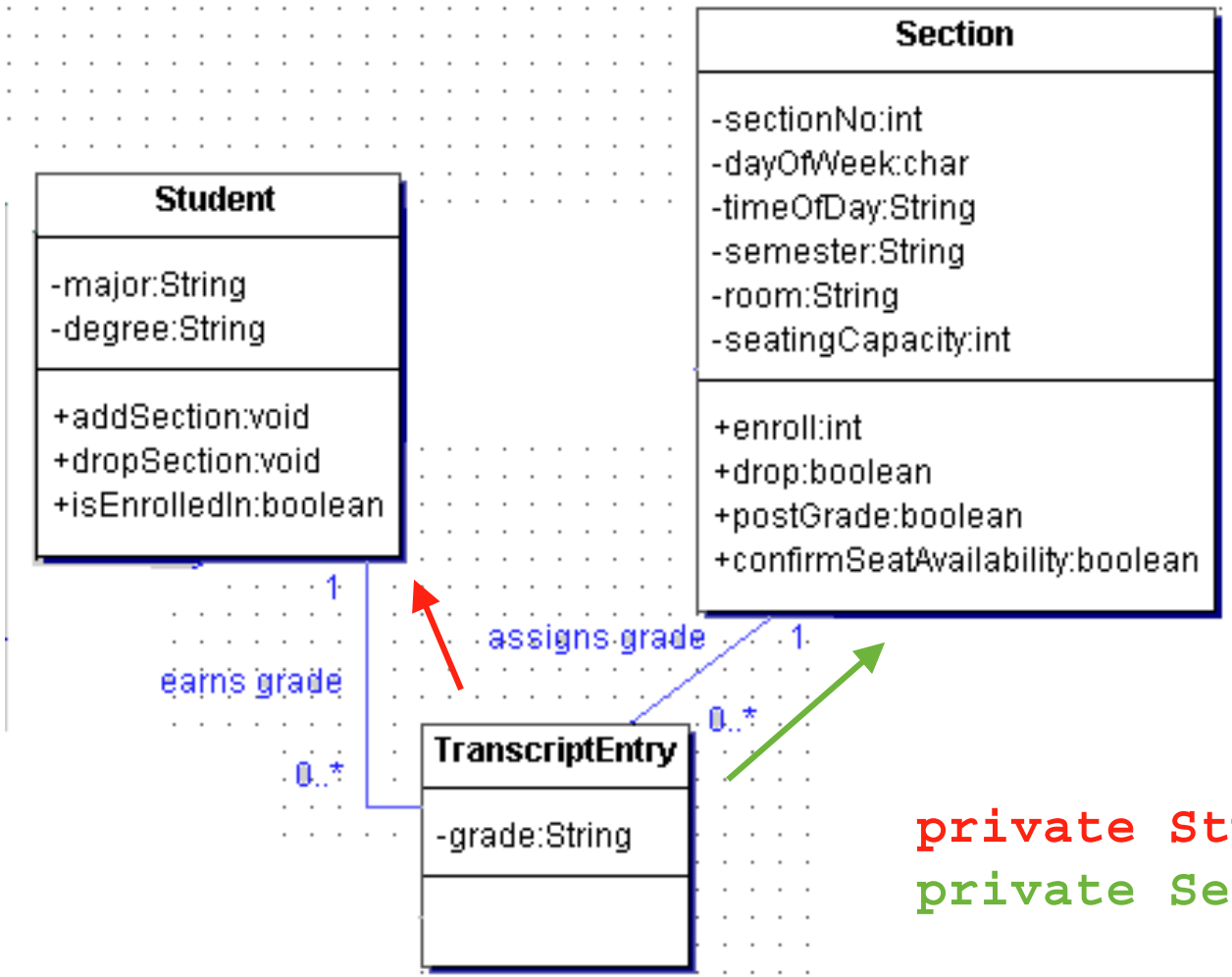
UML → Java: Relation conversion by *Together* – some things are done automatically …

- Together distinguishes between a relation's *client* and *supplier* classes.
- This holds for associations, aggregations, and compositions.
- In the client Java class, it automatically generates an attribute for the relation:

```
public class Student { // ...
  /**
   * @label maintains
   * @clientCardinality 1
   * @supplierCardinality 1
   */
private Transcript lnkTranscript;}
```

- The programmer can rename this attribute with a more useful name like `transcript`.

UML ➔ Java: Relation conversion by *Together* – … but this may not be enough



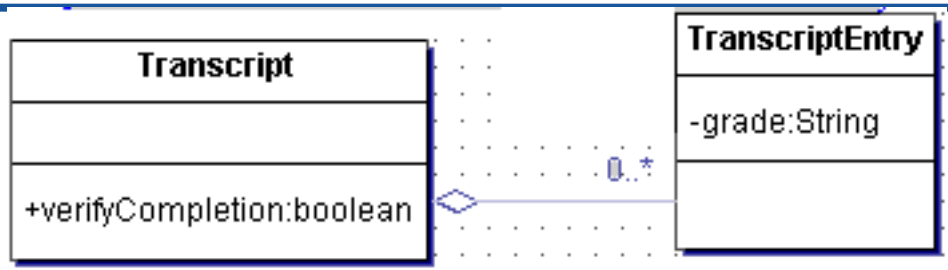
Each of these 2 associations needs at least an attribute in `TranscriptEntry`.

If `TranscriptEntry` is the *client* class of the association in *Together*, the following two attributes are generated automatically:

```
private Student lnkStudent;
private Section lnkSection;
```

But: We may also need access (➔ attributes) in the reverse direction!

UML ➔ Java: Relation conversion by *Together* – take care with *x*-to-many relations!

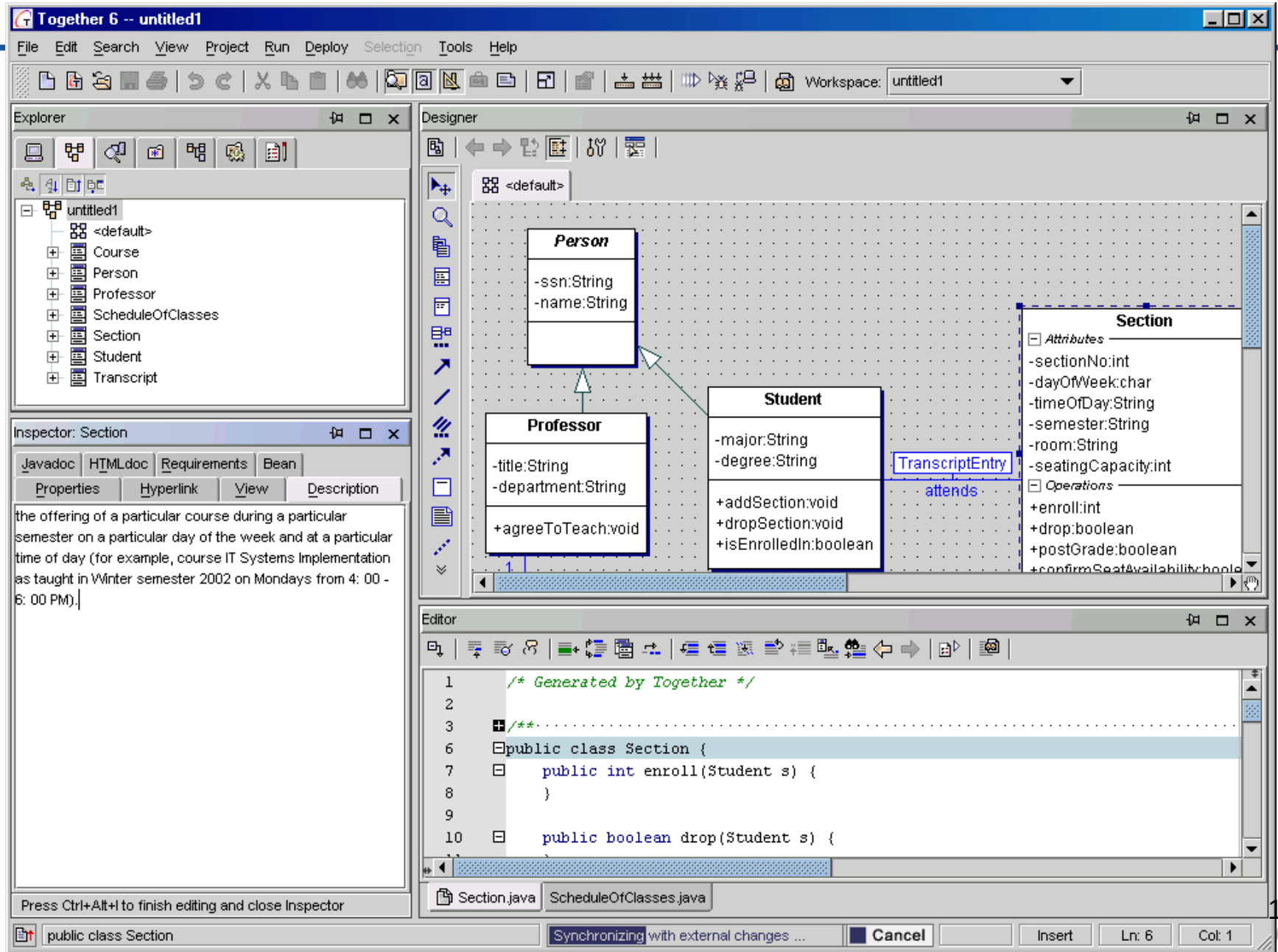


- To generate this direction of the part-whole relation, the Transcript class (the whole) must be the aggregation's client
- *Together* then produces:

```
public class Transcript {
    // ...
    private TranscriptEntry lnkTranscriptEntry; }
```

- This needs to be corrected by hand in two respects:
  - The type of this attribute must be a collection!
  - A link from the TranscriptEntry may need to be added

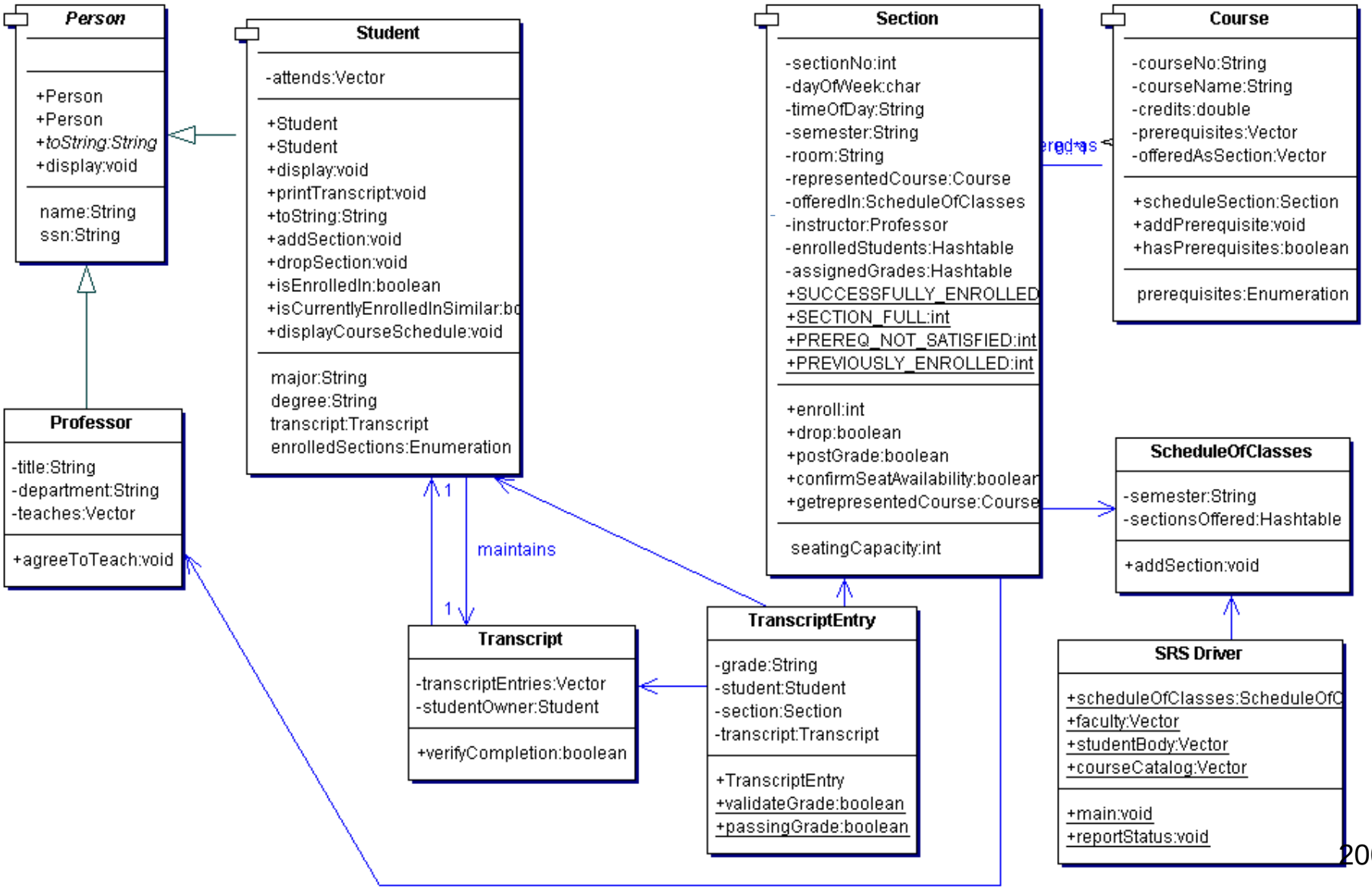
Using the data dictionary in *Together*



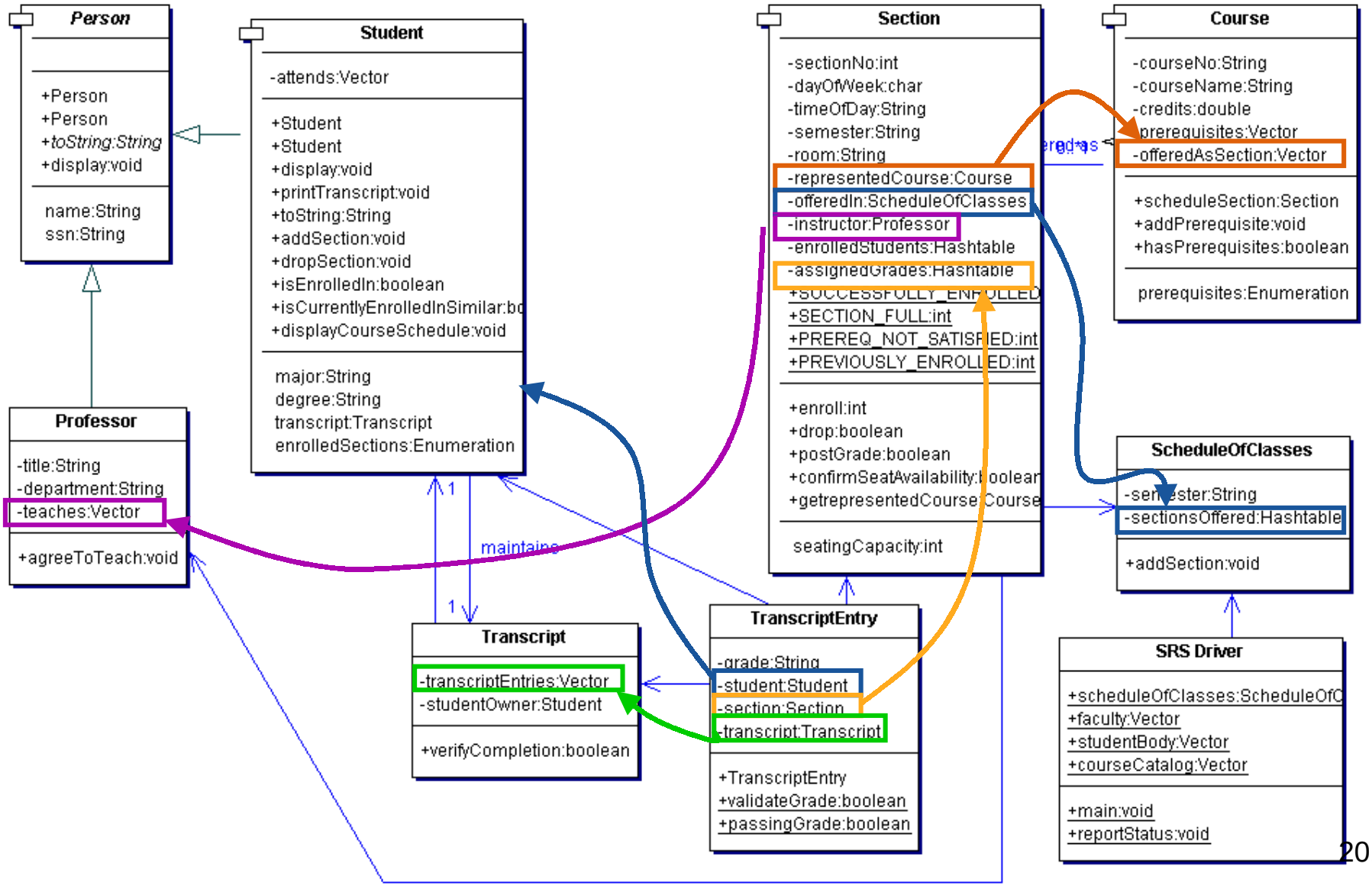
Specifics of *Together*

- Operations are shown without parentheses and without their signature
- „+“ indicates a public attribute/operation, „-“ a private one
- Association classes are only partially supported.
- Constructor(s) are shown like other operations.
- Structural relations have a *„supplier“* and a *„client“* class.
- Whenever there is a „get“ and/or a „set“ method associated with an attribute, Together treats this attribute as a *„property“*, and the operation(s) are not shown in the class diagram.

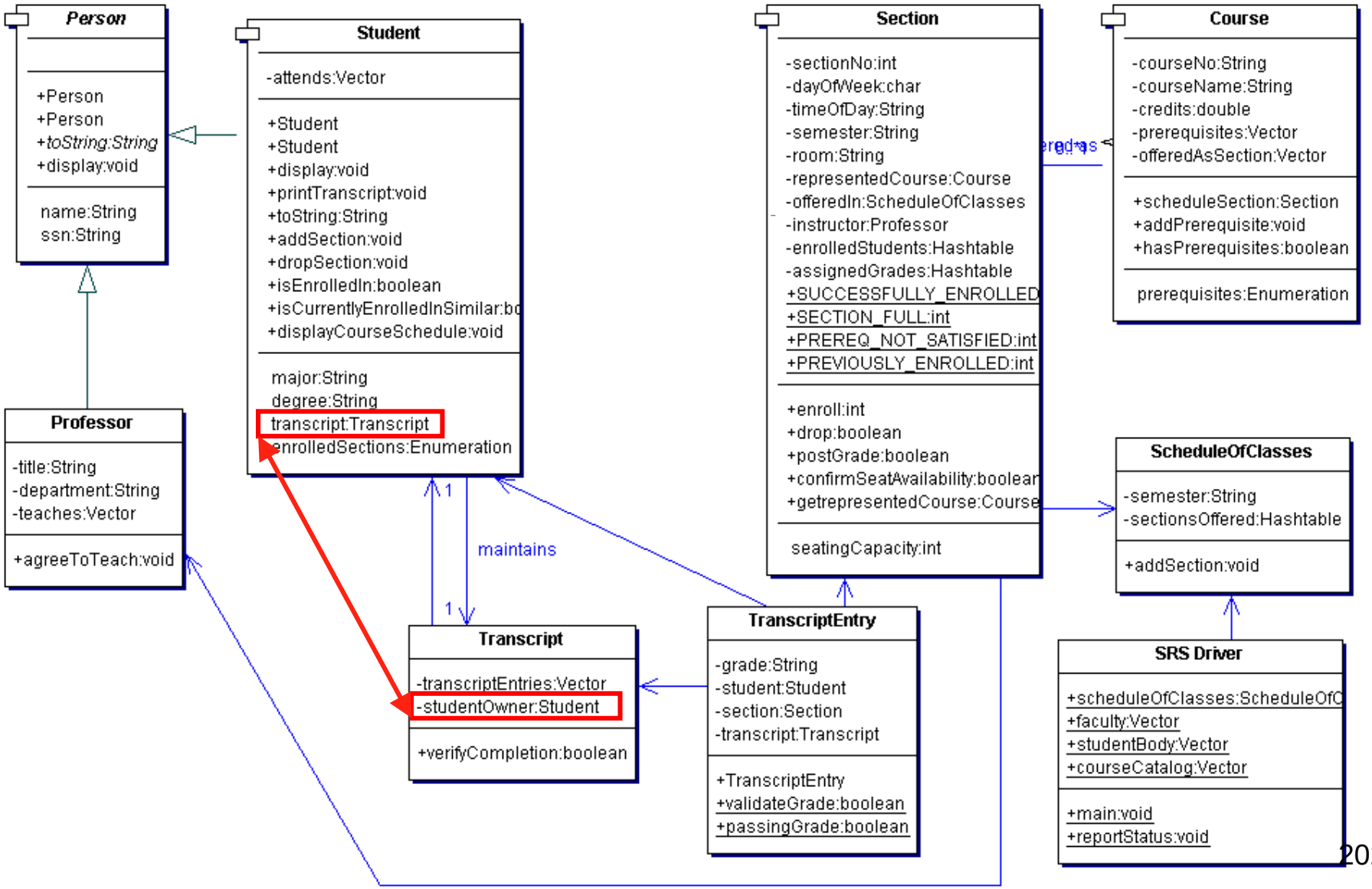
The *Together* UML class diagram C: the one that corresponds to the SRS Java source code



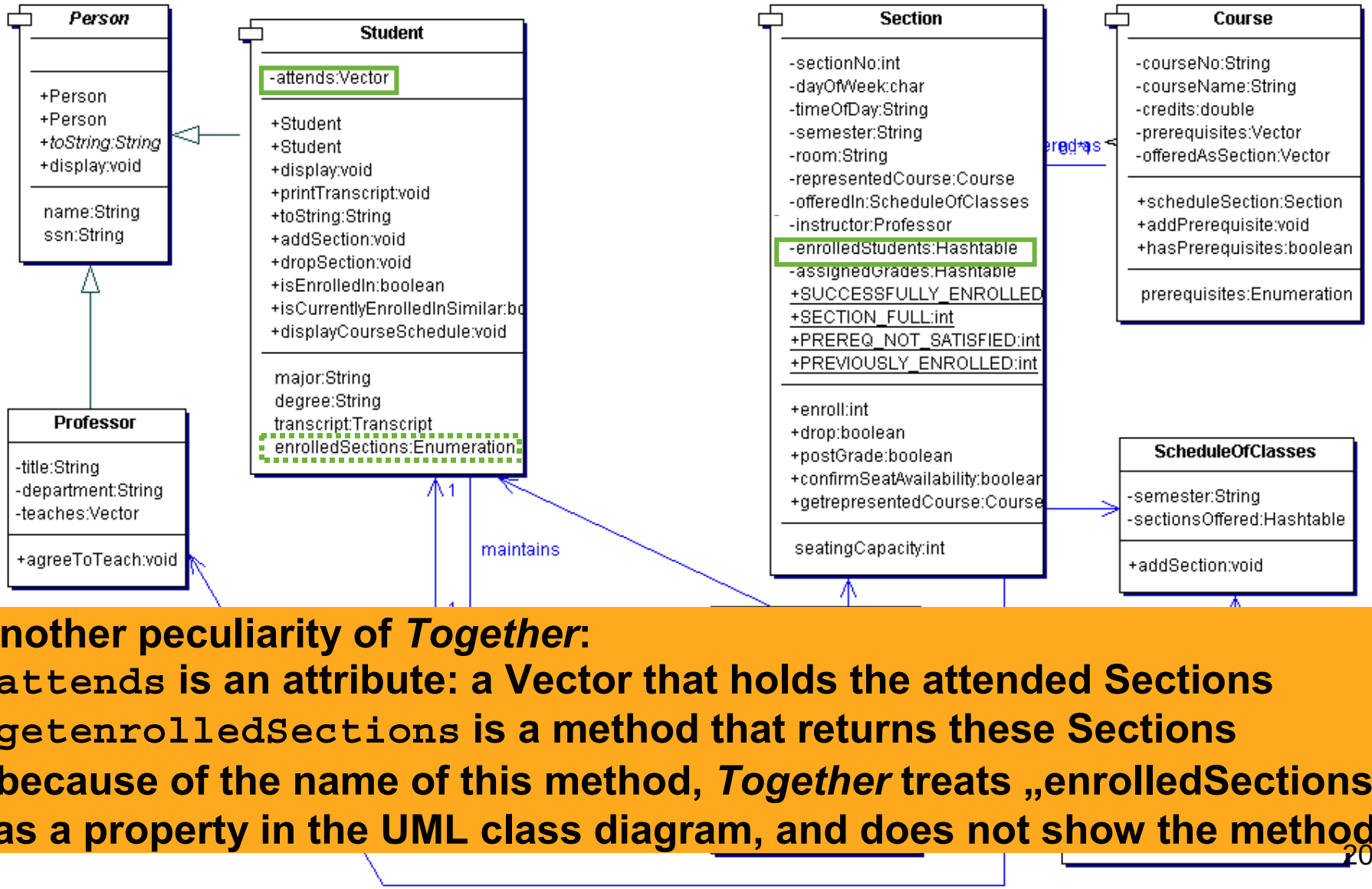
The *Together* UML class diagram C: One-to-many associations are shown as (1 or) 2 attributes & 1 arrow



The *Together* UML class diagram C: One-to-one associations are shown as (1 or) 2 attributes & 2 arrows



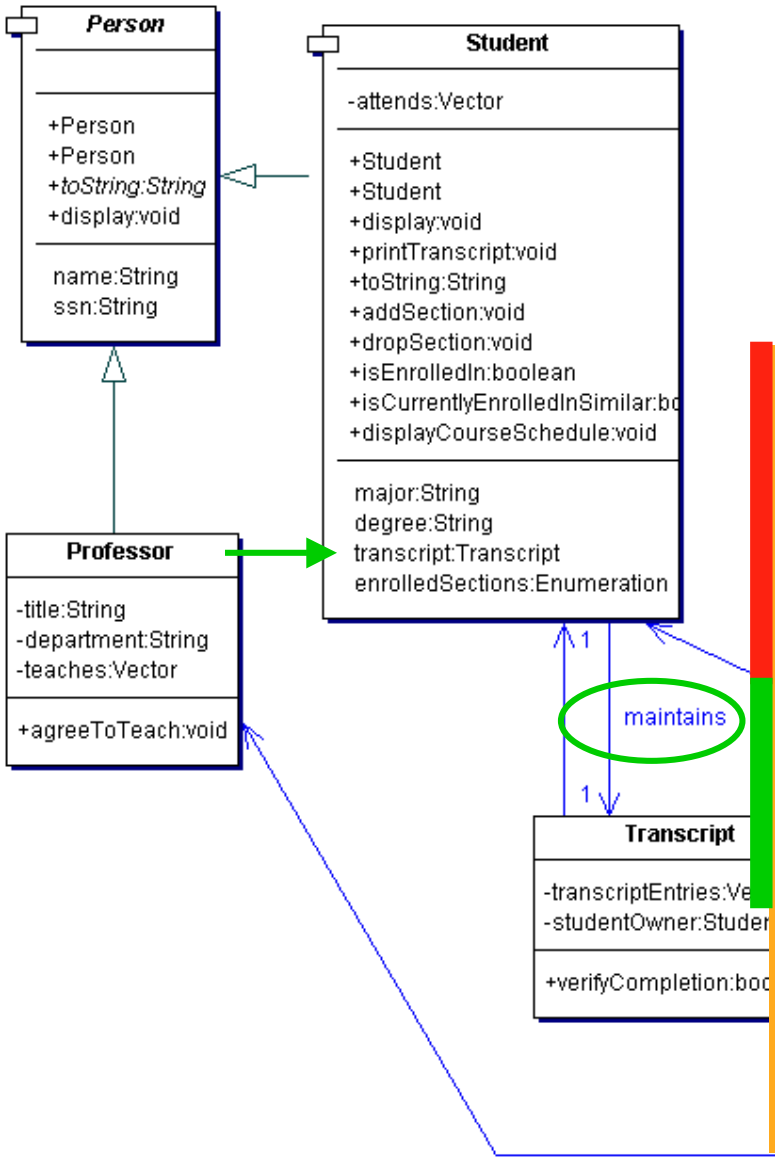
The *Together* UML class diagram C: Many-to-many associations are shown as 2 attributes & no arrows

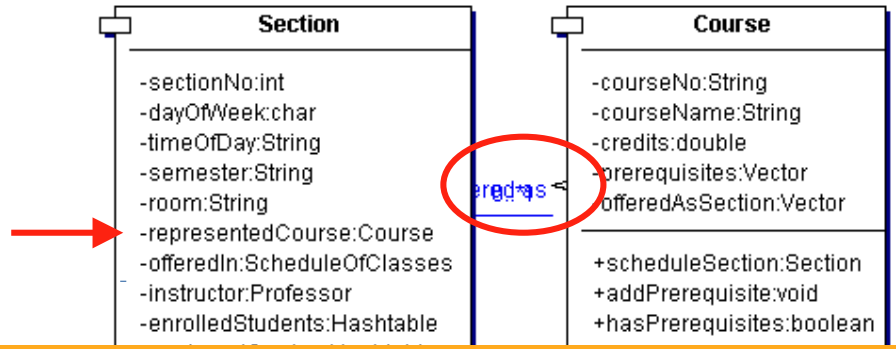


Another peculiarity of *Together*:

- **`attends` is an attribute: a Vector that holds the attended Sections**
- **`getenrolledSections` is a method that returns these Sections**
- **because of the name of this method, *Together* treats „enrolledSections“ as a property in the UML class diagram, and does not show the method**

Effect of renaming link variables generated by *Together* in the implementation of an associations' client class





`Section` was the client class of the association `offered as`.

➔ Its implementation obtained a variable `Course lnkCourse`.

This was renamed `representedCourse`.

The association label remains in the diagram!

Similarly, `Student` was the client class of the association `maintains`. The variable `Transcript lnkTranscript` was renamed `transcript`.

All other generated link variables were replaced by collection-type variables (e.g., in `Student: Section lnkSection` ➔ `Vector attends`). These disappear from the diagram