
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS401 Modern Programming Practices (MPP)
Professor Renuka Mohanraj

Lecture 2: Associations, Modeling Relationships with UML

Diversifying Self-Referral Relationship to the World of Objects

Wholeness Statement

In the real world, objects have relationships. These manifested relationships appear in many different ways. When these relationships are modeled in UML, those that reflect a permanent relationship are called *associations*.

At the most fundamental level every object in existence is made out of the same essence – and is therefore (in a way) related to everything. An intellectual analysis or model of all these relationships is generally not practical.

A direct experience of the underlying reality of all of manifest creation and our relationship with all of nature is a result of our practice of Transcendental Meditation.

Overview

- **Types of relationships between classes: association, dependency, inheritance**
- Techniques for discovering associations
  - Identify verb phrases
  - Create an association matrix
- Aspects of associations
  - Unidirectional and bidirectional associations
  - Aggregation
  - Composition
  - Reflexive associations
  - Association classes
  - Dependency
  - Association “decorations”: name, roles, multiplicities

Relationships Between Classes

In the OO paradigm, there are three fundamental types of relationships that can exist between classes

- Association
- Dependency
- Inheritance (discussed in Lesson 3)

(continued)

		
Association from A to B	Dependency from A to B	A inherits from B

Associations

1. "Customer *has an* Account"
2. Permanent relationship
3. Association from A to B implies A keeps a reference to B
4. Association from A to B implies it is possible to navigate from A to B at runtime

Dependencies

1. "RightTriangle *uses* Math" (see example)
2. Temporary relationship
3. Dependency from A to B implies A does *not* keep a reference to B

Demo: lesson2.lecture.dependencyexample

Two Examples

- Association

```
public class Customer {
    private Account checkingAccount;
    public void createNewAccount() {
        checkingAccount = new Account();
    }
}
```

- Dependency

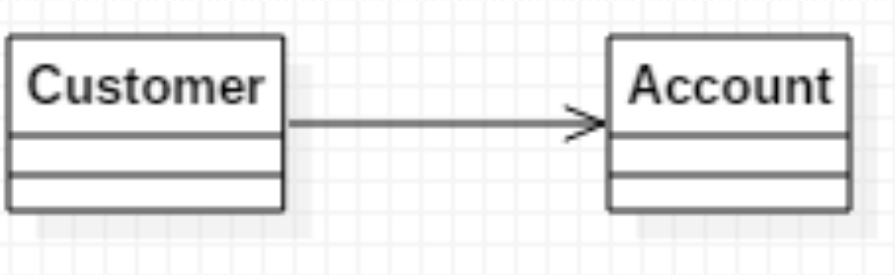
```
public class RightTriangle {
    public static double computeHypotenuseLength(double base, double height) {
        return Math.sqrt(base * base + height * height);
    }
}
```

Dependency Example

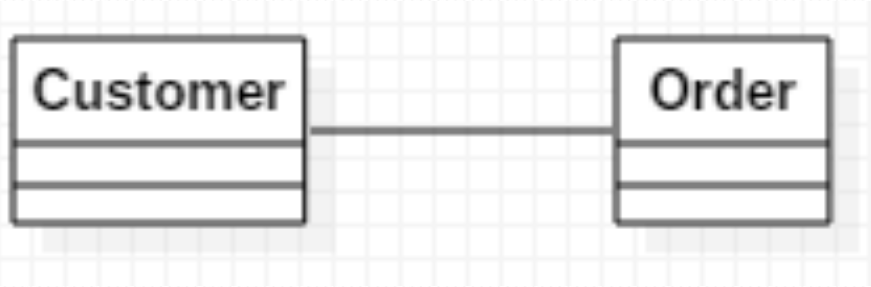
```
public class Email {
public void sendEmail(String subject, String message){
System.out.println(“Subject : ” + subject + “\n Message : “  + message);
}
}

public class Person {
public void greetFriend(Email ob){
   ob.sendEmail("Hello", "Hello my friend :)");
 }
 }
```

One-way and Two-way Associations

- Sometimes should be able to navigate from A to B but not from B to A. This is a *one-way* or *uni-directional* association

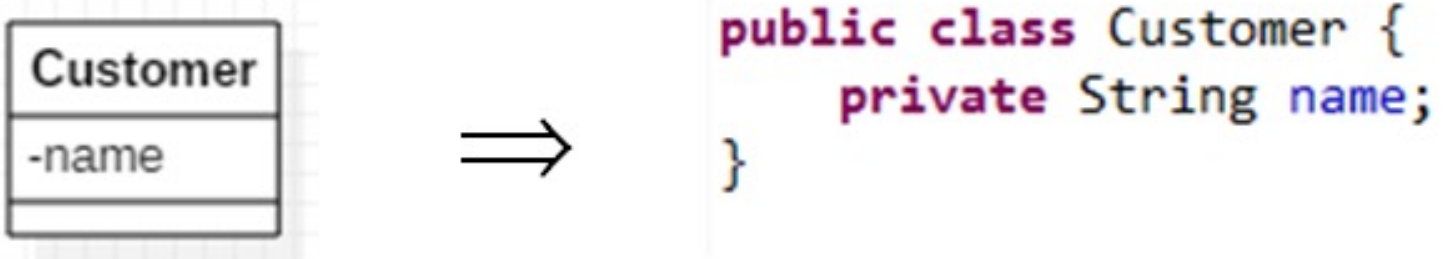


- Sometimes should be able to navigate from A to B and also from B to A. This is a *two-way* or *bi-directional* association

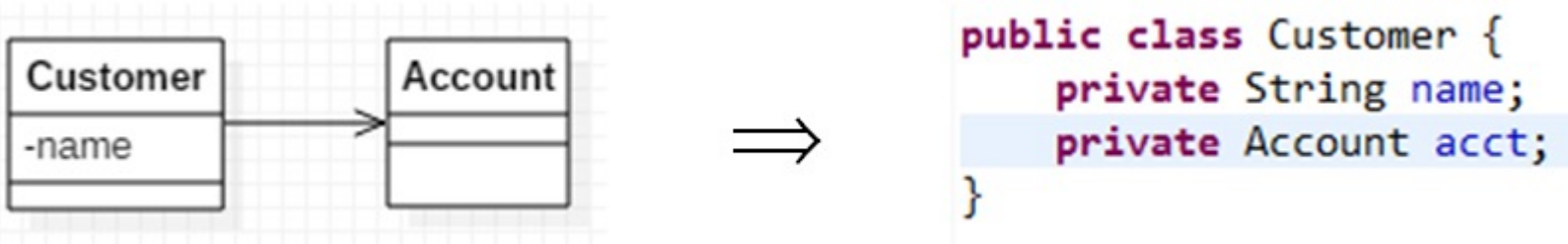


Properties as Attributes or Associations

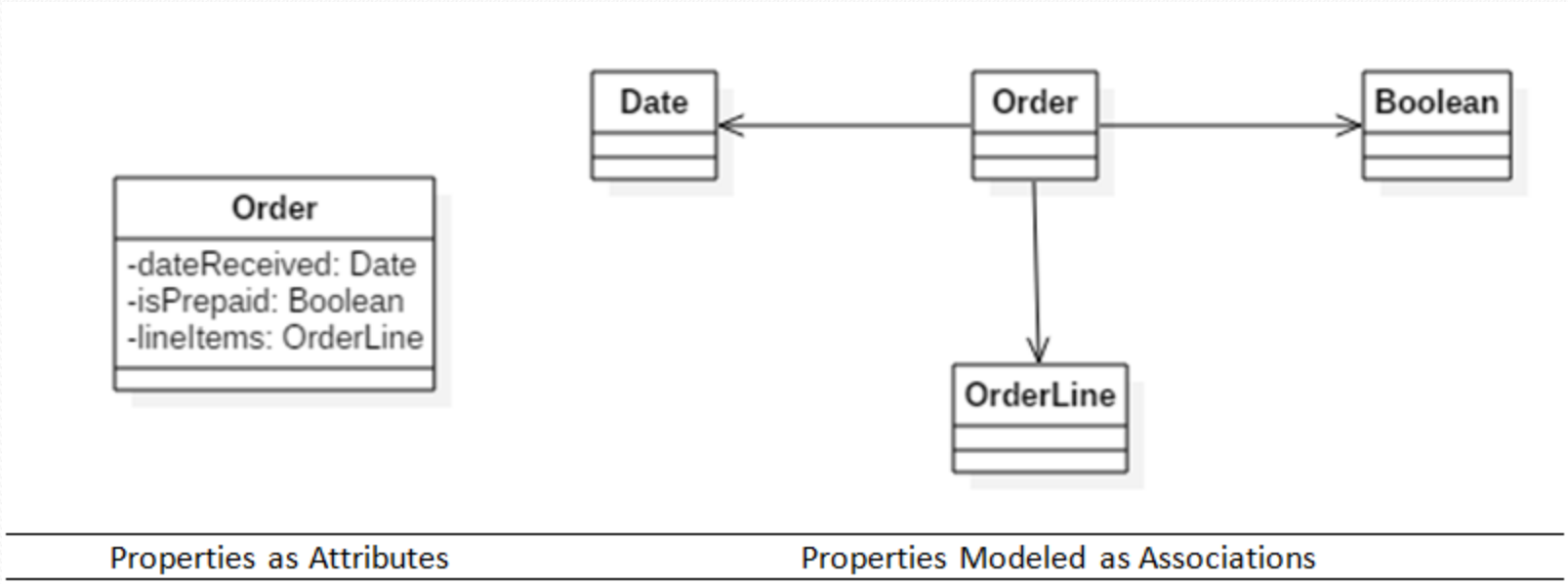
- An *attribute* of a UML class indicates a variable that stores data, like `name` in `Customer`. A UML attribute is implemented in Java code as a *field* or *instance variable*.



- An association from one class to another is also implemented in Java code as a field or instance variable, like the association from `Customer` to `Account`



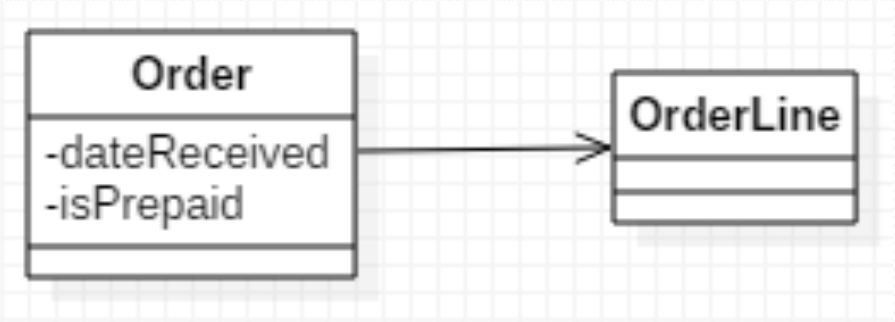
- In the language of UML, both `acct` and `name` are called *properties* of `Customer`. The property `acct` is modeled as an association; the property `name` is modeled as an attribute.
- In UML, it is always possible to model properties either as attributes or as associations:



Which way is best – properties as *attributes* or *associations*?

- When a property has an internal structure, with its own properties, model it as an association; otherwise, as an attribute.

Example: In the previous example, an `Order` has many `OrderLines`, and each `OrderLine` has its own internal structure (cost, quantity, etc). So the `OrderLine` property should be modeled as an association; the others, as attributes



IMPORTANT: When a property is modeled as an association, it is not mentioned as one of the attributes of the class. Here, `Order` displays two attributes, one association. All three would appear as *fields* in a Java class implementation

Overview

- Types of relationships between classes: association, dependency, inheritance
- **Techniques for discovering associations**
  - **Identify verb phrases**
  - **Create an association matrix**
- Aspects of associations
  - Unidirectional and bidirectional associations
  - Aggregation
  - Composition
  - Reflexive associations
  - Association classes
  - Dependency
  - Association “decorations”: name, roles, multiplicities

Name of an Association Is a Verb

Examples

- Customer *has* an Account
- Professor *advises* a Student
- Student *enrolls in* a Section

Strategy:

- Discover associations by finding verbs and verb phrases in the problem statement.
- *Optional*: Track the relationships in an *Association Matrix*

Problem Description for the SRS

We have been asked to develop an automated Student Registration System (SRS) for the university. This system will enable students to register online for courses each semester, as well as track their progress toward completion of their degree.

When a student first enrolls at the university, he/she uses the SRS to create a plan of study that lists the courses he/she plans on taking to satisfy a particular degree program, and chooses a faculty advisor. The SRS will verify whether or not the proposed plan of study satisfies the requirements of the degree that the student is seeking.

Once a plan of study has been established, then, during the registration period preceding each semester, students are able to view the schedule of classes online and choose whichever classes they wish to attend, indicating the preferred section (day of the week and time of day) if the class is offered by more than one professor.

The SRS will verify whether or not the student has satisfied the necessary prerequisites for each requested course by referring to the student's online transcript of courses completed and grades received (the student may review his/her transcript online at any time).

Assuming that (a) the prerequisites for the requested course(s) are satisfied, (b) the course(s) meet(s) one of the student's plan of study requirements, and (c) there is room available in each of the class(es), the student is enrolled in the class(es).

If (a) and (b) are satisfied, but (c) is not, the student is placed on a first-come, first-served wait list. If a class/section that he/she was previously waitlisted for becomes available (either because some other student has dropped the class or because the seating capacity for the class has been increased), the student is automatically enrolled in the waitlisted class, and an email message to that effect is sent to the student. It is the student's responsibility to drop the class if it is no longer desired; otherwise, he/she will be billed for the course.

Students may drop a class up to the end of the first week of the semester in which the class is being taught.

Association Matrix

	Section	Course	Plan of Study	Professor	Student	Transcript
Section		**instance of**		is taught by		
Course						
Plan of Study						
Professor						
Student						
Transcript						

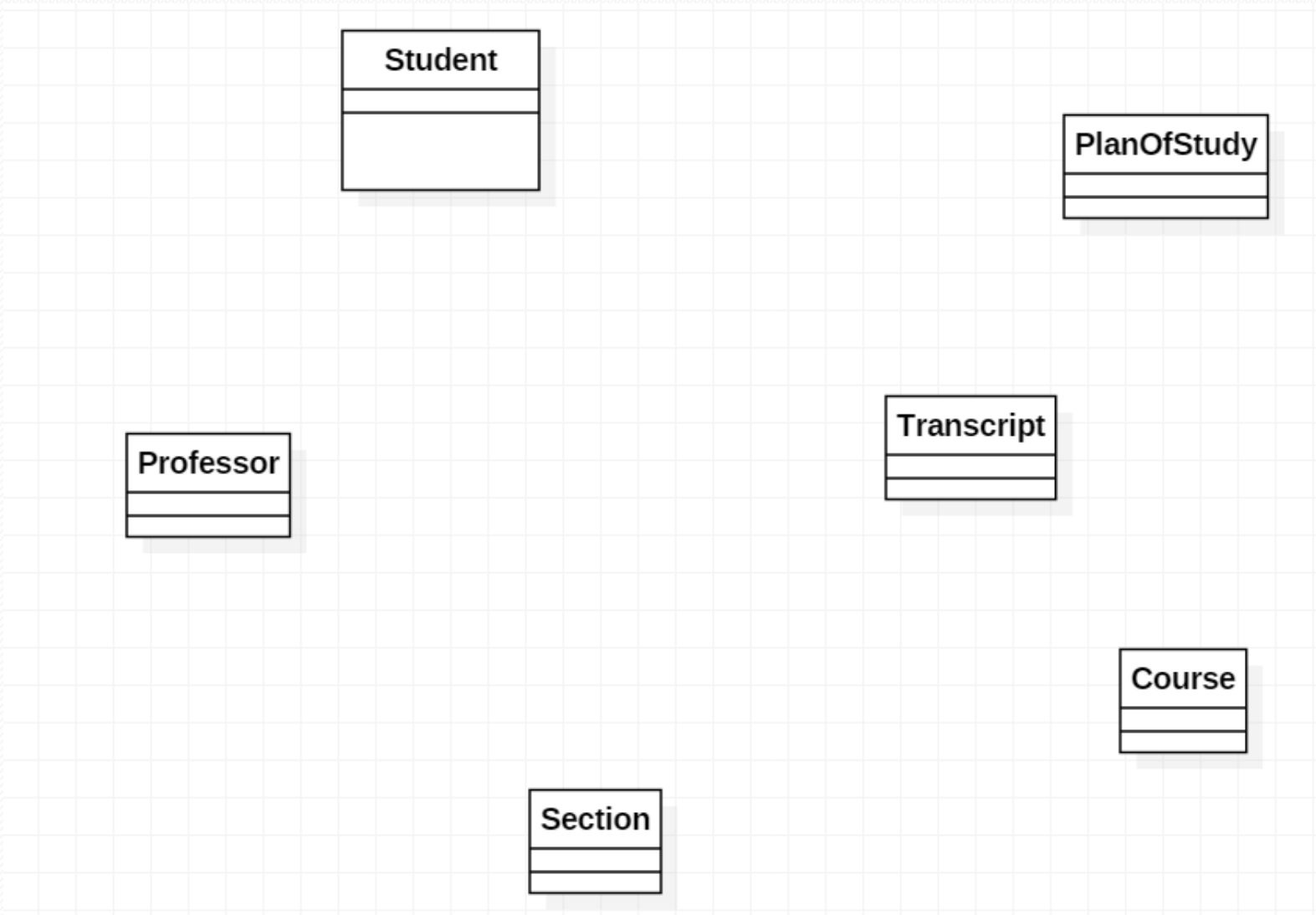
	Section	Course	Plan of Study	Professor	Student	Transcript
Section		instance of		is taught by		included in
Course	offered as	prerequisite for	is called for by			
Plan of Study		calls for			observed by	
Professor	teaches				advises; teaches	
Student	registered for; waitlisted for; has previously taken	plans to take	observes	is advised by; studies under		owns
Transcript	includes	includes			belongs to	

Exercise 2.1: Specifying Associations in the SRS

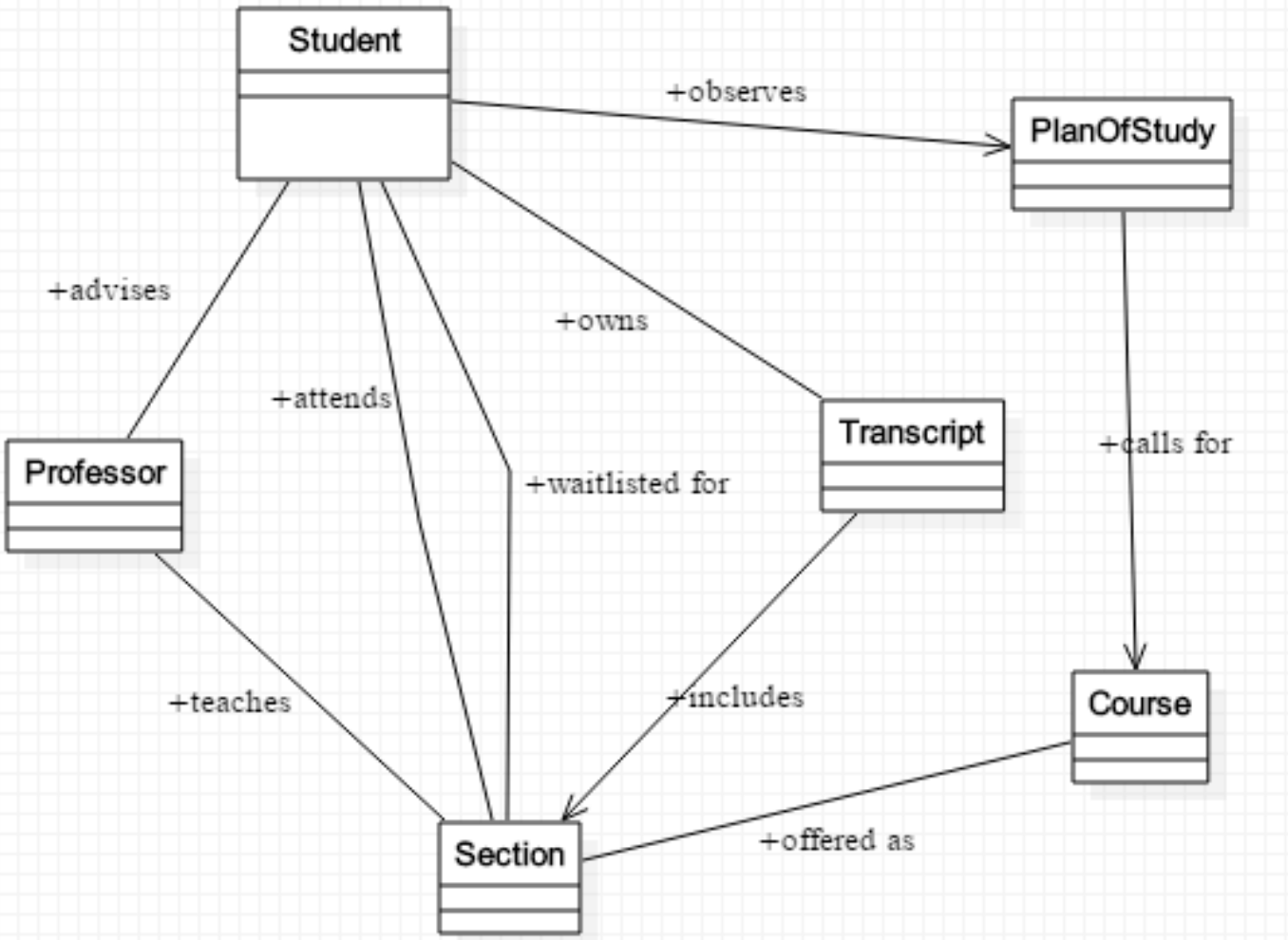
In your small group create a diagram with all the classes and their associations – show association names

Final List of Classes

Course (rather than Class)
PlanOfStudy
Professor
Section
Student
Transcript


Student
PlanOfStudy
Professor
Transcript
Course
Section

Student Registration System



Main Point 1

Building a software system using OO principles involves an *analysis* step in which the problem is analyzed and broken into pieces as objects are discovered. The pieces are then refined and put together – in a step of *synthesis* – to give a picture of a unified system. This step of synthesis happens in part through the identification of relationships between classes, represented by *associations*.

This phenomenon is a characteristic of all knowledge – it arises through a combination of analysis and synthesis.

Quiz 1

1. Which one of the following indicates temporary relationship between the classes?

   a. Inheritance

   b. Association

   c. Dependency

Overview

- Types of relationships between classes: association, dependency, inheritance
- Techniques for discovering associations
  - Identify verb phrases
  - Create an association matrix
- **Aspects of associations**
  - **Unidirectional and bidirectional associations**
  - **Aggregation**
  - **Composition**
  - **Reflexive associations**
  - **Association classes**
  - **Dependency**
  - **Association "decorations": name, roles, multiplicities**

Association - Unidirectional

- Unidirectional
  - Objects of a class have a reference to an object of another class.
  - The association can be given a descriptive name (a verb), often with a direction indicator [some UML tools do not support direction indicators]


```
public class Customer {
    private ShoppingCart cart;
}
```

```
public class ShoppingCart{

}
```

Association:
Multiplicities

- UML supports a variety of multiplicities

1	one (mandatory)
3	three (exactly)
*	many
0..*	zero or more (optional)
1..*	one or more
0..1	zero or one (optional)

Determining Multiplicity

Ask:

For a given instance of a class A and association involving A, B, how many instances of B must/may be associated with this instance of A?

Optionality

Is the association required? If not, association will be "zero or more"; otherwise, it will be "one or more"

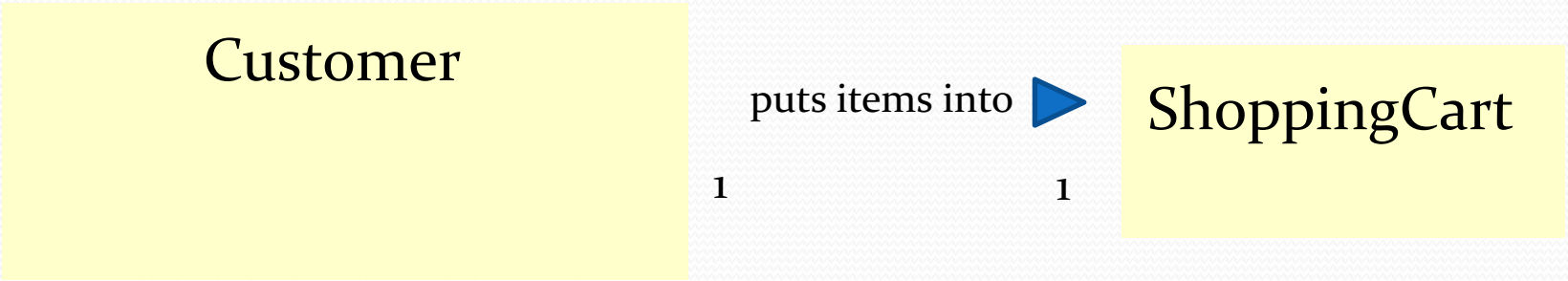
Cardinality

How many instances are associated with a given instance? Could be 1:1, 1:2, 1:3, 1..* (for example)

UML combines both ideas in the concept of multiplicity

Unidirectional with Multiplicities

A unidirectional association may be 1-0..1, 1-1, or one-many.



One-one Multiplicity:

- Associated with each Customer, there is exactly one ShoppingCart
- Associated with each ShoppingCart, there is exactly one Customer
- It is possible to navigate from a Customer to his ShoppingCart, but not from ShoppingCart to Customer.
- *Maintaining the relationship* means:
  - when new Customer object is created, it is equipped with a ShoppingCart
  - it is not possible to add a second ShoppingCart to a Customer object
  - it is not possible to create a ShoppingCart on its own; it must be created as a property of a Customer object.

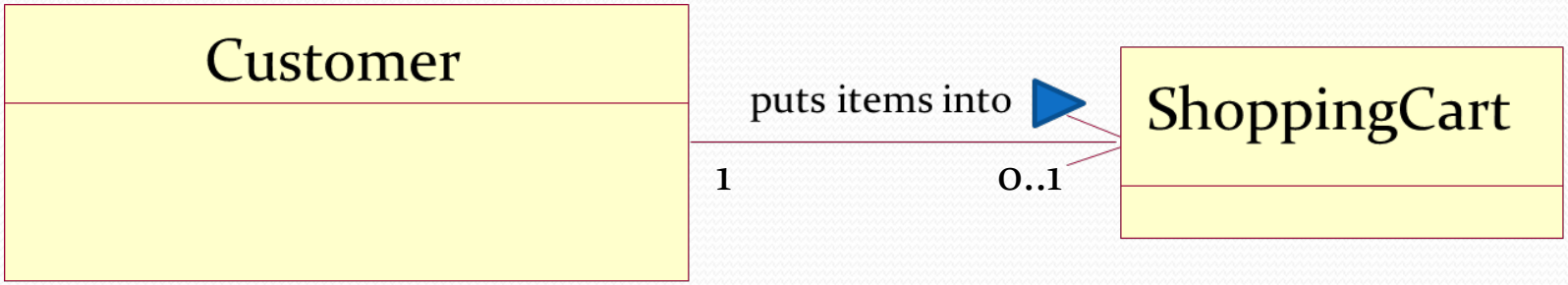
Example: 1-1 and Unidirectional

```
public class Customer {
    private String name;
    private ShoppingCart cart;
    public Customer(String name) {
        this.name = name;
        cart = new ShoppingCart();
    }
    public String getName() {
        return name;
    }
    public ShoppingCart getCart() {
        return cart;
    }
}
```

```
public class ShoppingCart {
    private List<Item> items;
    public void addItem(Item item) {
        items.add(item);
    }
    //package level
    ShoppingCart() {
        items = new ArrayList<Item>();
    }

    public List<Item> getItems() {
        return items;
    }
}
```

See lesson2.lecture.unidirectional.oneone

Unidirectional with Multiplicities



One to zero..one Multiplicity:

- Associated with each Customer, there is zero or one ShoppingCart
- Associated with each ShoppingCart, there is exactly one Customer
- It is possible to navigate from a Customer to his ShoppingCart, but not from ShoppingCart to Customer.
- *Maintaining the relationship* means:
  - Customer class has a shopping cart variable that *may be* populated
  - it is not possible to add a second ShoppingCart to a Customer object
  - it is not possible to create a ShoppingCart without an owning Customer object.

Example: Unidirectional, One to Zero..One(Easy)

```
public class Customer {
        private String name;
        private ShoppingCart cart;
        public Customer(String name) {
                this.name = name;
        }
        public String getName() {
                return name;
        }
        public void addCart() {
          if(cart == null)
          cart = new ShoppingCart();
        }
        public ShoppingCart getCart() {
                return cart;
        }
}
```

```
public class ShoppingCart {
        private List<Item> items;

        public void addItem(Item item) {
                items.add(item);
        }

        /** package level constructor */
        ShoppingCart() {
        items = new ArrayList<Item>();
        }

        public List<Item> getItems() {
                return items;
        }
}
```

See: lesson2.lecture.unidirectional.oneToZeroOneEasy

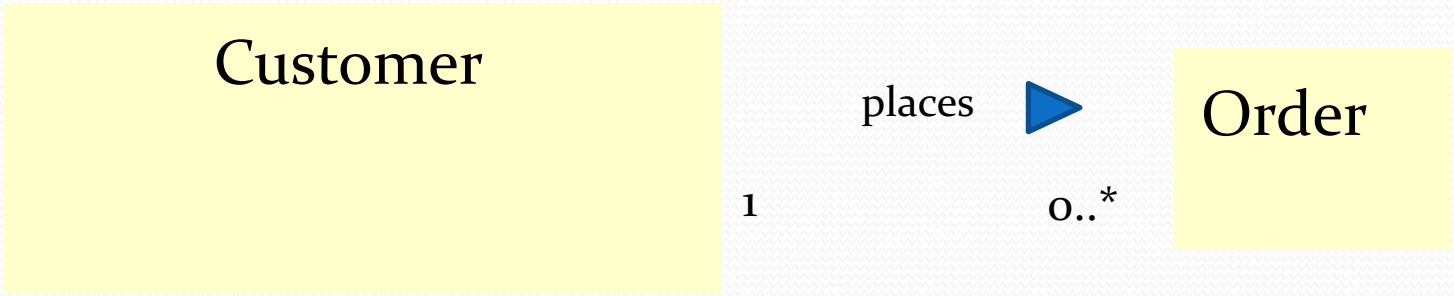
Example: Unidirectional, One to Zero..One

```java
public class Customer {
    private String name;
    private ShoppingCart cart;
    public Customer(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setCart(ShoppingCart cart)
        this.cart = cart;
    }
    public ShoppingCart getCart() {
        return cart;
    }
}
```

```java
public class ShoppingCart {
    private List<Item> items;

    public void addItem(Item item) {
        items.add(item);
    }

    /** Use a factory method for construction */
    private ShoppingCart(Customer cust) {
        items = new ArrayList<Item>();
        cust.setCart(this);
    }
    public static ShoppingCart newShoppingCart(
                                        Customer cust) {
        if (cust == null)
            throw new NullPointerException(
                                    "Null customer");
        return new ShoppingCart(cust);
    }
    public List<Item> getItems() {
        return items;
    }
}
```

See:
lesson2.lecture.unidirectional.oneToZeroOne

Unidirectional with Multiplicities



One-many Multiplicity:

- Associated with each Customer, there are zero or more Orders
- A Customer object maintains a collection of Order objects.
- Associated with each Order, there is exactly one Customer
- It is possible to navigate from a Customer to any of his Orders, but not from an Order to the owning Customer.
- *Maintaining the relationship* means:
  - when new Customer object is created, it is equipped with a (possibly empty) collection of Orders
  - it is not possible to create an Order object independent of a Customer; each new Order object must belong to the collection of Orders for some Customer.

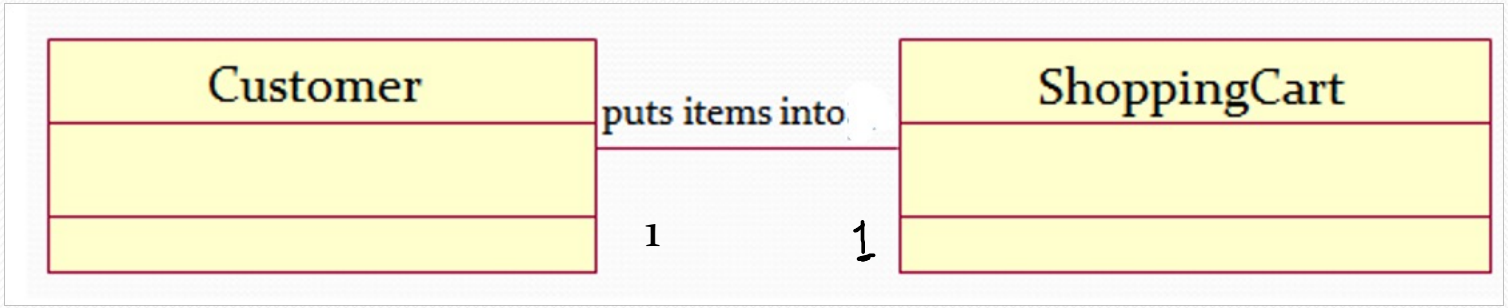
Example: Unidirectional, 1-many

```java
public class Customer {
    private String name;
    private List<Order> orders;
    public Customer(String name) {
        this.name = name;
        orders = new ArrayList<Order>();
    }
    public Order addOrder(LocalDate dateOfOrder) {
        Order ord = new Order(dateOfOrder);
        orders.add(ord);
        return ord;
    }
    public String getName() {
        return name;
    }
    public List<Order> getOrders() {
        return orders;
    }
}
```

```java
public class Order {
    private LocalDate orderDate;
    private List<Item> items;
    //package level access
    Order(LocalDate orderDate) {
        this.orderDate = orderDate;
        items = new ArrayList<Item>();
    }
    public void addItem(String name){
        items.add(new Item(name));
    }
    @Override
    public String toString() {
        return orderDate + ": " +
                items.toString();
    }
}
```

\lesson2\lecture\unidirectional\onemany

Bidirectional with Multiplicities



One-one Multiplicity:

- Associated with each Customer, there is exactly one ShoppingCart, and Customer contains a reference to its ShoppingCart
- Associated with each ShoppingCart, there is exactly one Customer, and ShoppingCart contains a reference to its owning Customer
- It is possible to navigate from a Customer to his ShoppingCart, and also from ShoppingCart to Customer.
- *Maintaining the relationship* implies:
  - when new Customer object is created, it is equipped with a ShoppingCart and a new ShoppingCart is equipped with a reference to its owning Customer
  - it is not possible to add a second ShoppingCart to a Customer object

(continued)

- Implementation Strategies:
  1. One of the classes *owns the relationship*. If Customer owns the relationship, ShoppingCart is created when Customer is created, and only then.

     OR

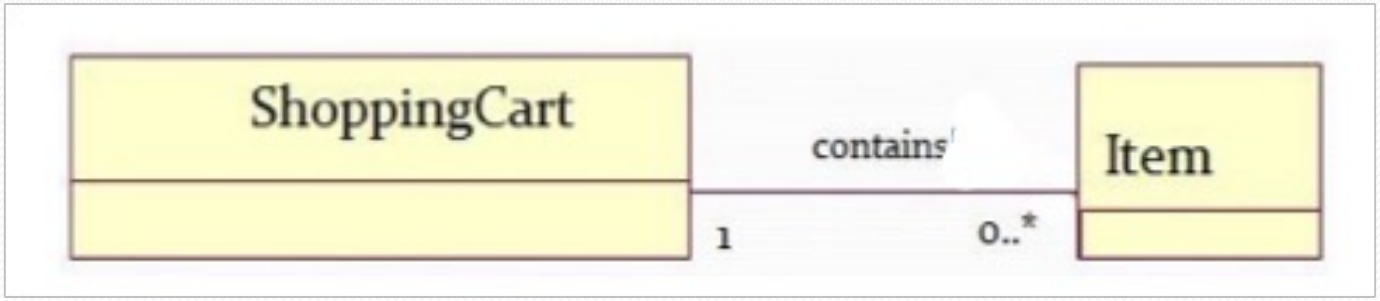
  2. The owner of the relationship is external to the two classes involved.

Examples of both of these are in these demos:

lesson2.lecture.bidir.onetoone

lesson2.lecture.birdir.onetoone_factory

Bidirectional with Multiplicities



One-many Multiplicity:

- ShoppingCart maintains a collection of zero or more Items and each Item contains a reference to its owning ShoppingCart.
- Items cannot be created outside the context of an owning ShoppingCart
- *Maintaining the relationship* implies:
  - when new ShoppingCart is created, it is equipped with a (possibly empty) collection of Items; when an Item is created, a reference to its owning ShoppingCart is stored in the Item.

(continued)

- Implementation Strategies:
  1. The "one" class (of one-many) *owns the relationship*. If ShoppingCart owns the relationship in above example, a list of Items is created when ShoppingCart is created, and each Item stores this instance of ShoppingCart.

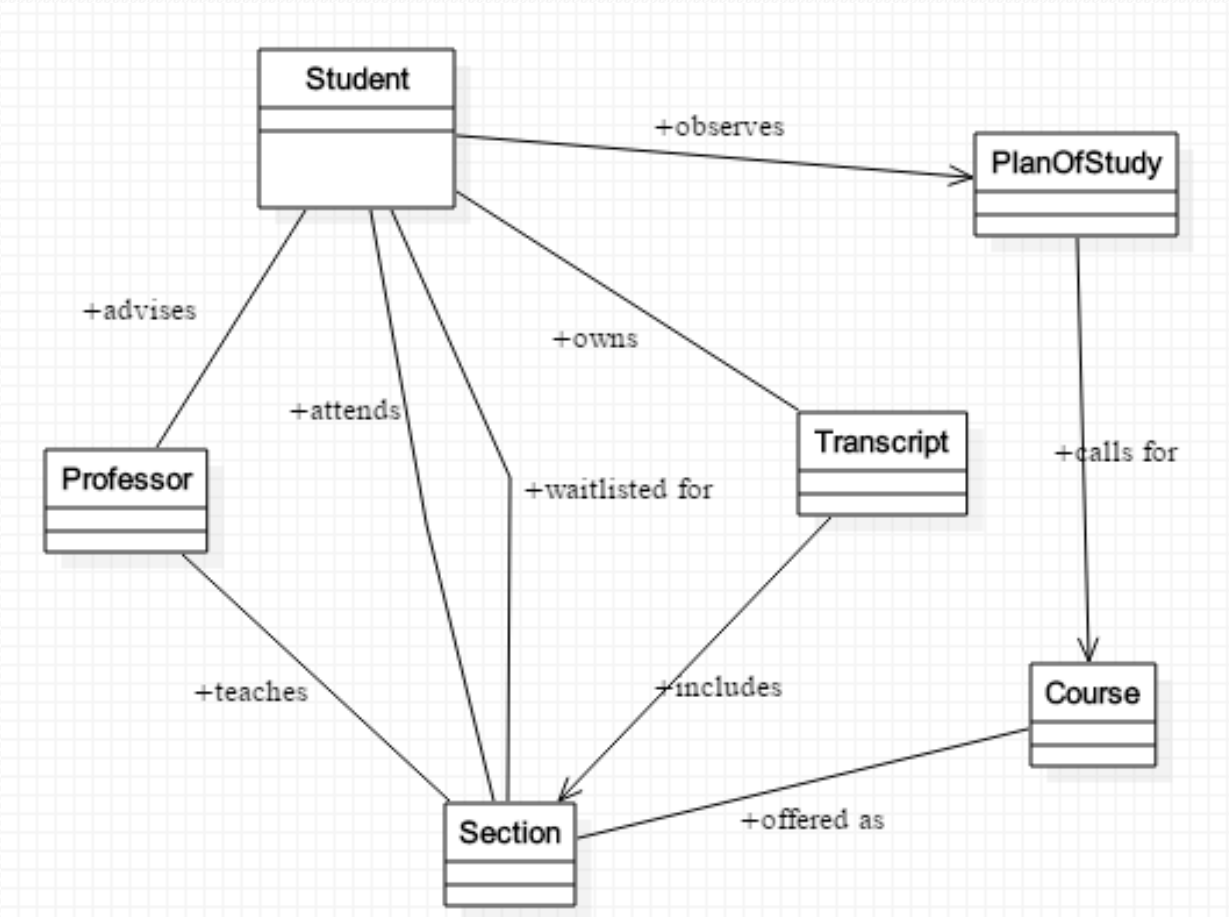
     OR

  2. The owner of the relationship is external to the two classes involved.

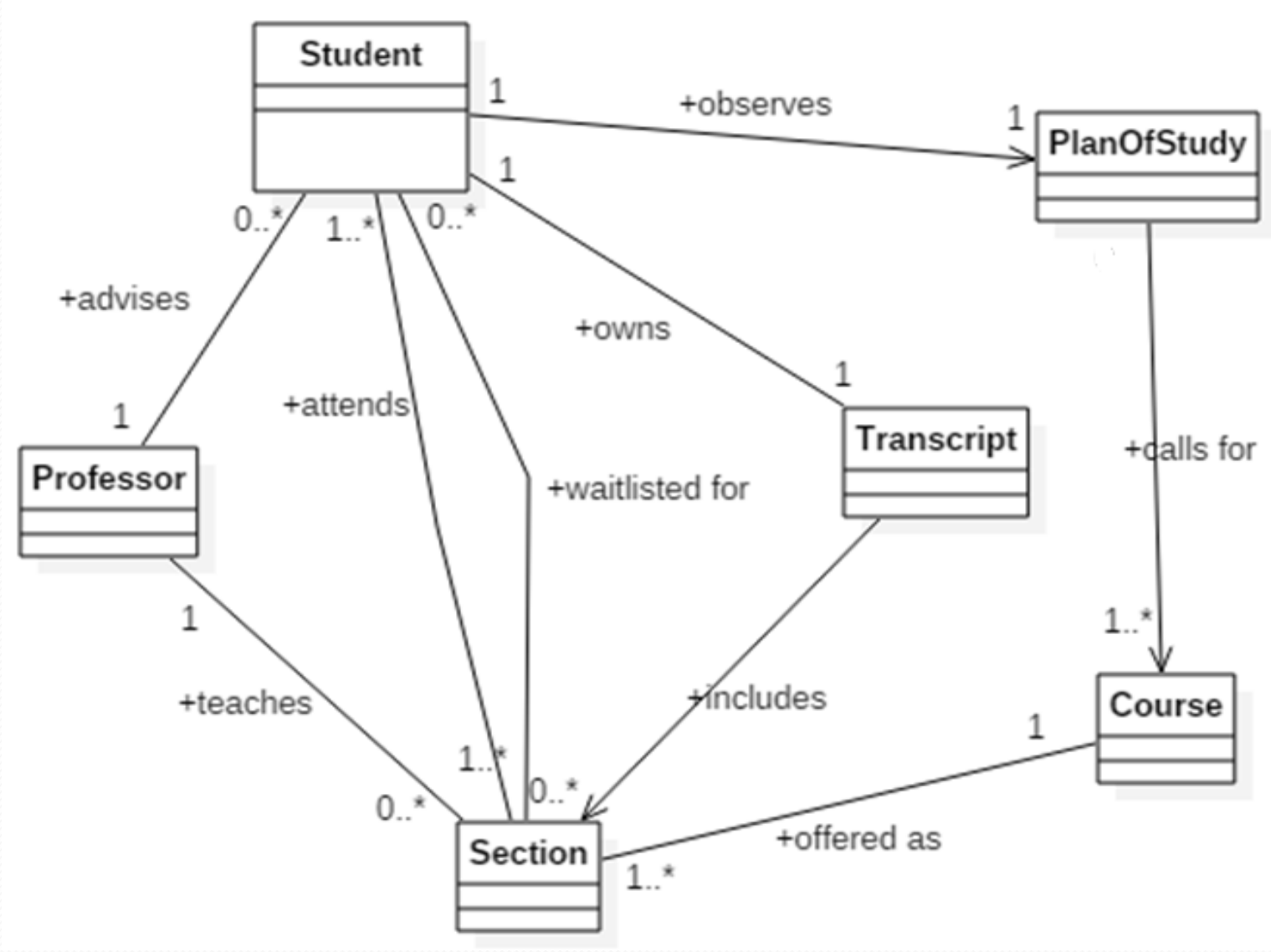
- See demo

  lesson2.lecture.bidir.onetomany

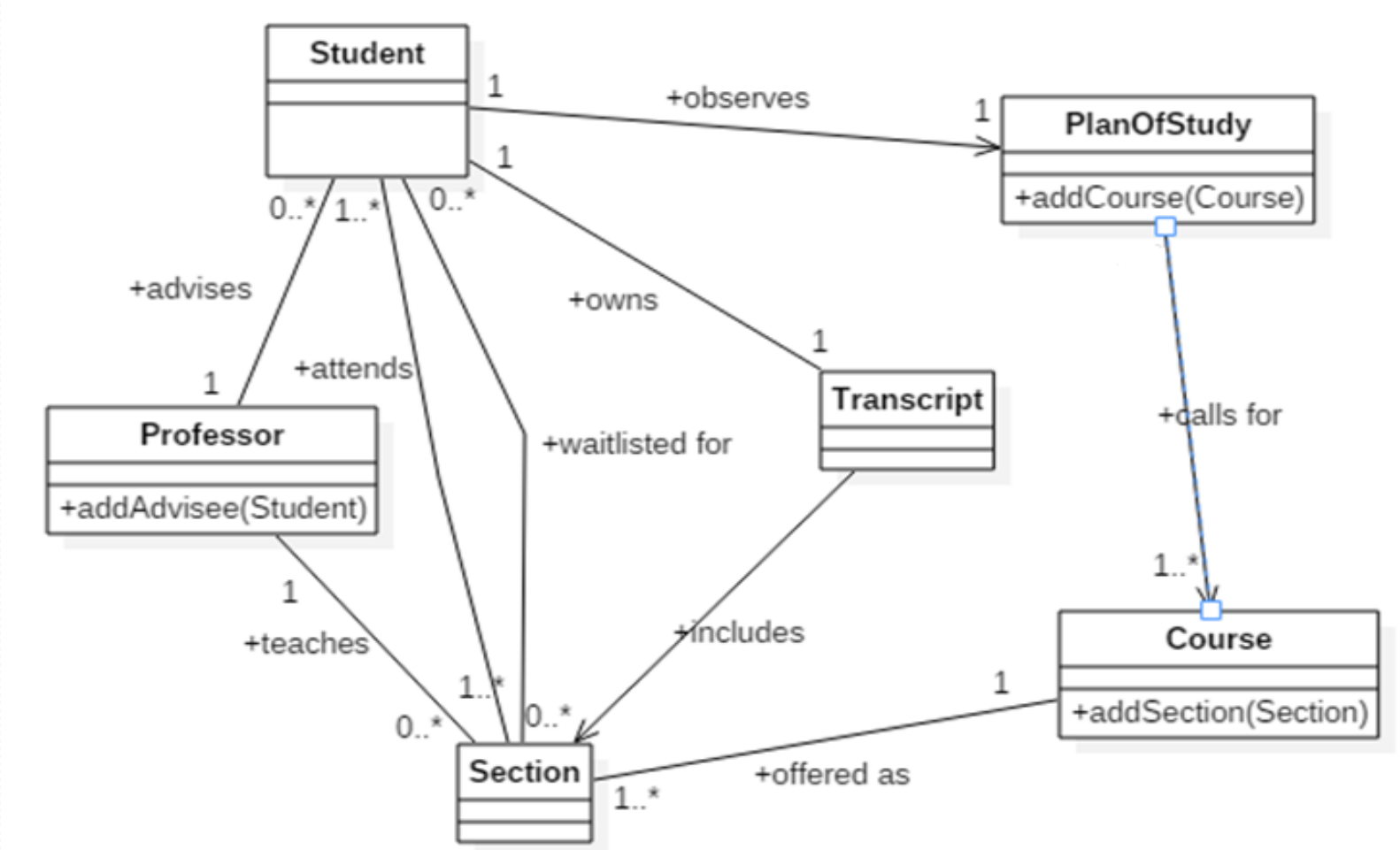
Exercise 2.2: Multiplicities in SRS?



Solution

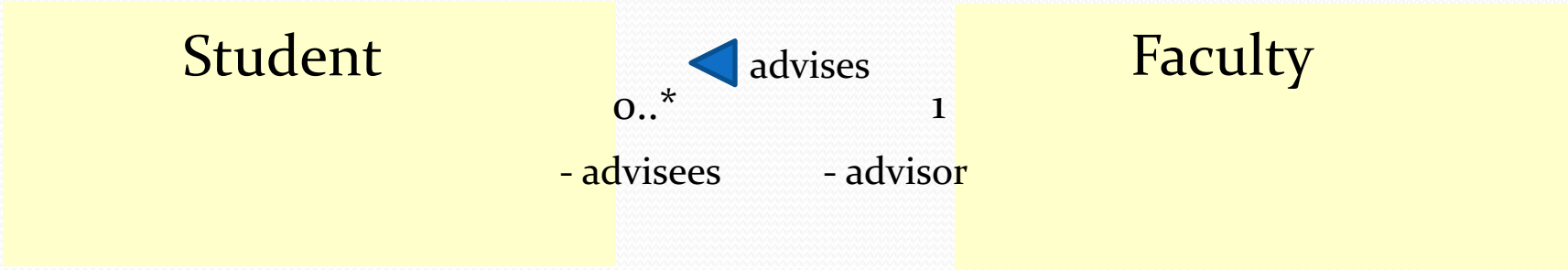

Student
PlanOfStudy
Professor
Transcript
Course
Section
+observes
+advises
+attends
+owns
+waitlisted for
+calls for
+teaches
+includes
+offered as
1
0..*
1..*

Associations Sometimes Suggest Operations



Association roles

A *role* is a (noun) description placed on either side of the association to indicate the role(s) each object plays in the relationship. Specifying roles is *optional*.




```
public class Student {
    private Faculty advisor;
}
```

```
public class Faculty {
    private List<Student> advisees;
}
```

Main Point 2

Associations model the relationships that can exist between concepts. Simple (one-way) associations are modeled using an *arrow*; two-way associations are modeled using a *line segment*.

The association can have a *name* for ease of reading, and additional symbols to indicate direction and multiplicity.

The ends of an association arrow can also specify association roles.

The "simplest" association is the relationship of pure consciousness to itself; this can also be modeled with an arrow from pure consciousness to itself.

Exercise 2.3

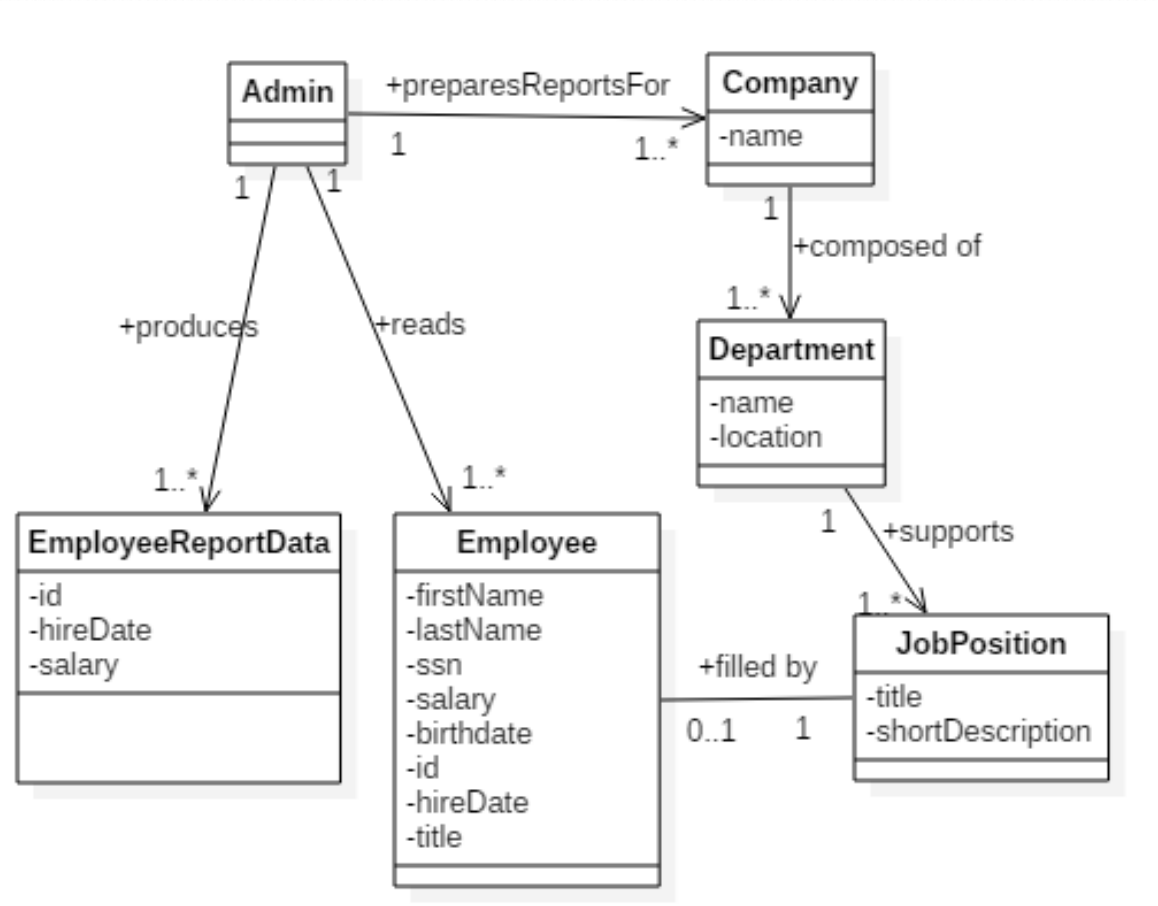
For the problem statement on the next slide, do the following:

- Work independently for 10 minutes to create the class diagram. Display the associations that are needed.
- Then review your diagrams for 10 minutes with your small group.

Exercise 2.3

A Human Resource (HR) department keeps track of employees for several companies, which it does using an Admin module in a software system. Each company has a name and is composed of one or more departments. A department has a name and a location. Each department has one or more job positions. A position has a title and a short description. Either a position is vacant or an employee is assigned to it. An employee has an id, title, first name, last name, birthdate, hire date, salary, and social security number . The HR department accesses the Admin module periodically to prepare a report for one of its companies. To do this, the company provides a list of all its employees. To prepare a report, the Admin module extracts from each employee record just the employee id, hire date, and salary, and then pieces together other information to create the final report. [Hint: Think of the HR dept as an *actor*.]

Solution

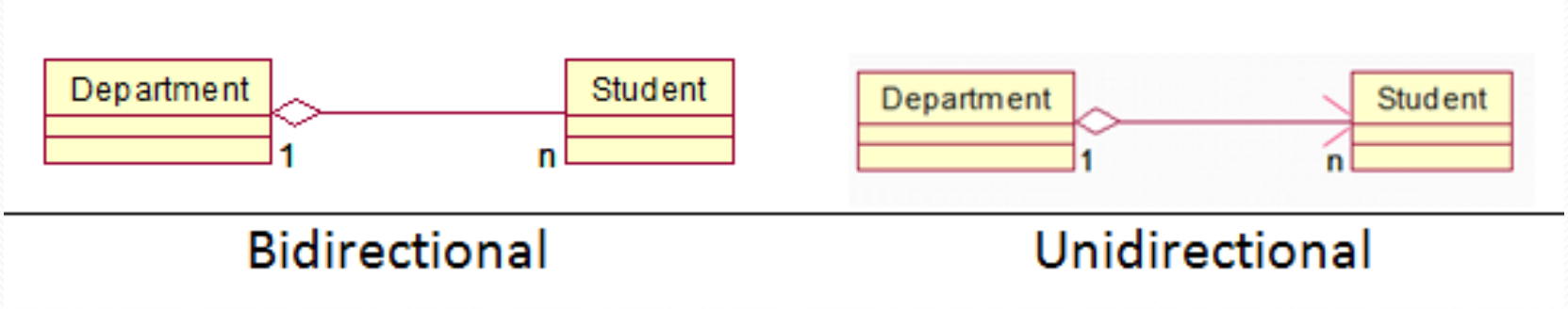

Admin
+preparesReportsFor
Company
-name
1
1..*
1
1
+composed of
1..*
+produces
+reads
Department
-name
-location
1..*
1..*
1
+supports
1..*
EmployeeReportData
-id
-hireDate
-salary
Employee
-firstName
-lastName
-ssn
-salary
-birthdate
-id
-hireDate
-title
+filled by
0..1
1
JobPosition
-title
-shortDescription

Aggregation

- Represents a 'whole-part' relationship
  - 'contains'
  - 'is part of'

← Association name is implied

- Code looks the same as an association.

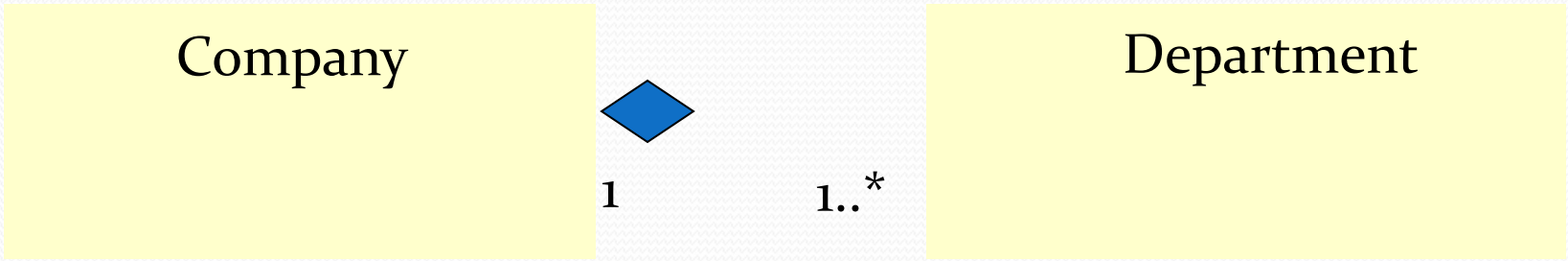



```
public class Department {
    private List<Student> students;
}
```

```
public class Student {
    private Department department;
}
```

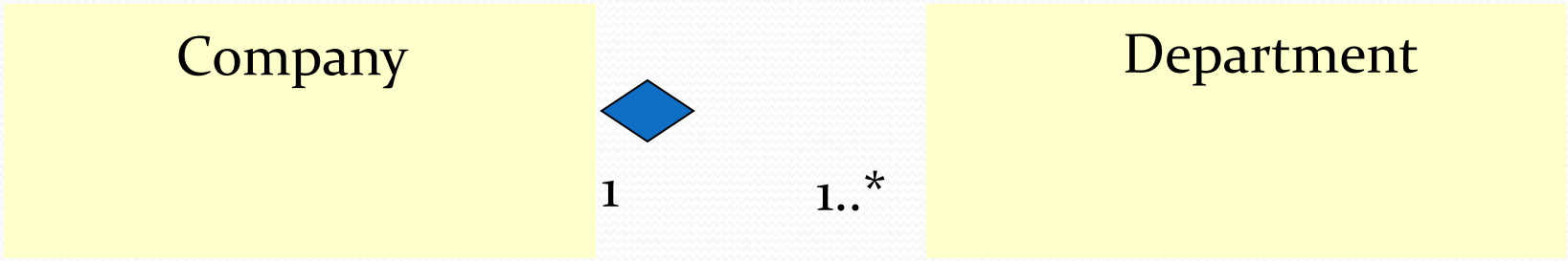
Composition

- Strong aggregation
  - 'whole-part' relationship – "is composed of"
  - If whole dies, parts also die.



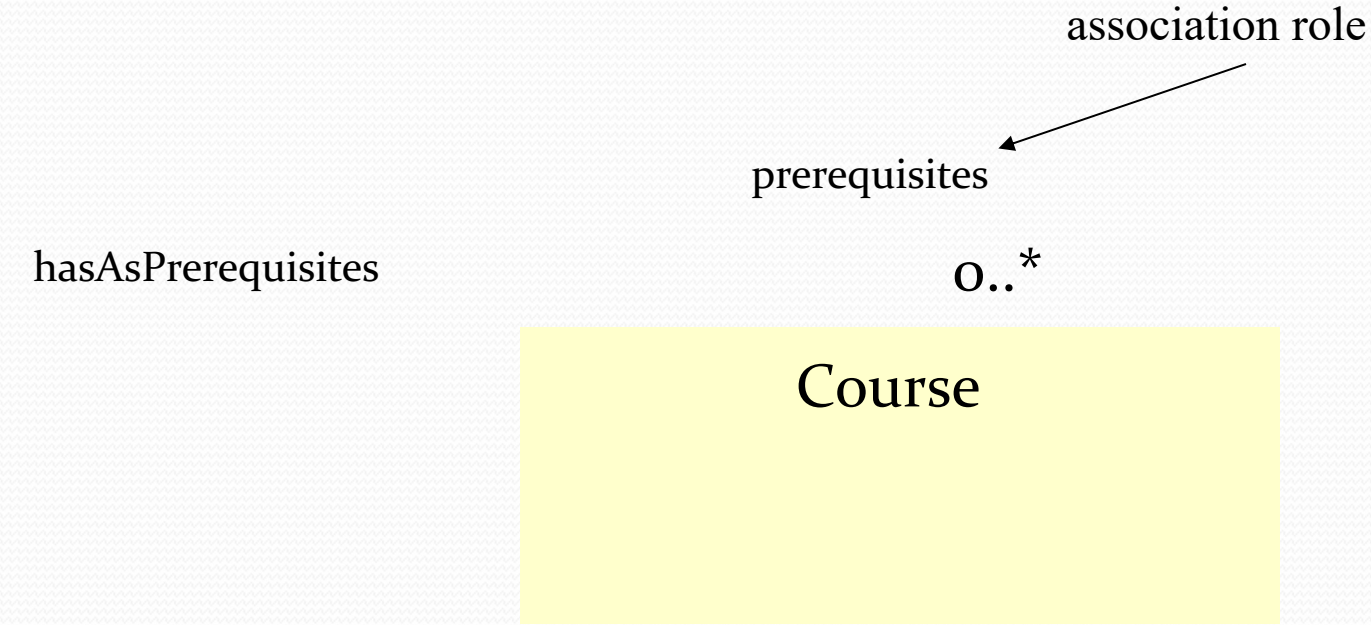
Composition

How to implement in code? There is no way (using Java) to ensure the composition relationship without using additional classes (beyond the two classes involved in the relationship).



Reflexive Association

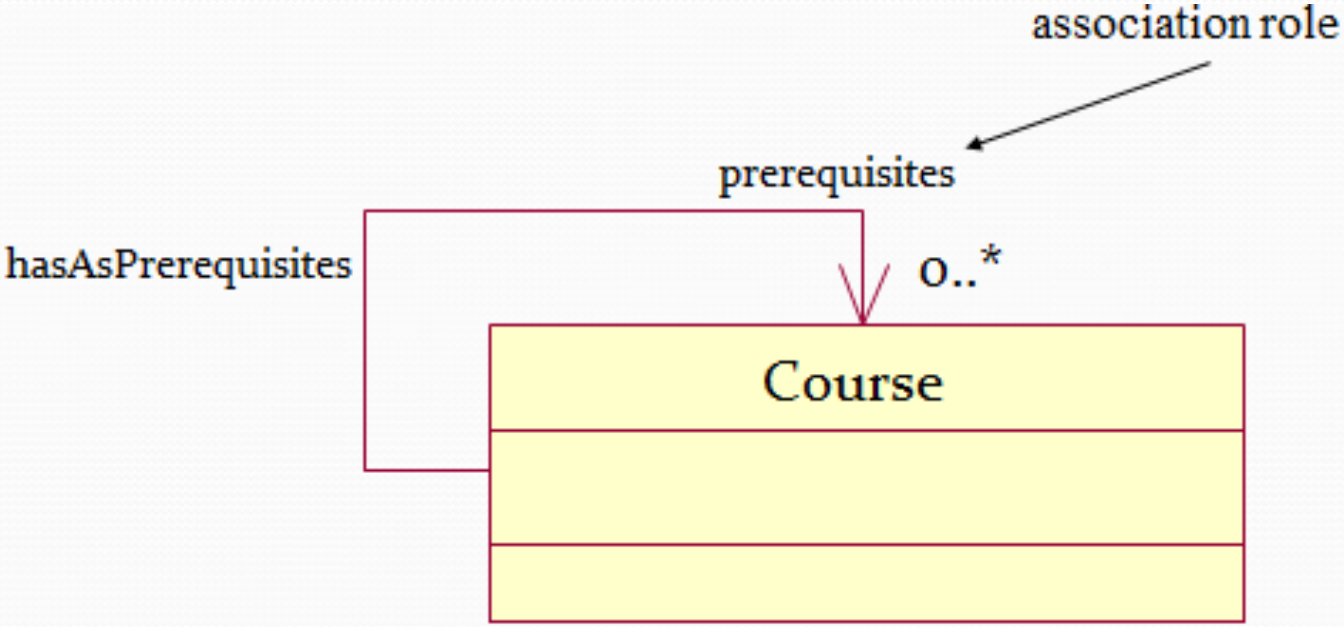
- Relationship between two or more objects of the same class.



How can this relationship be expressed in code?

Reflexive Association

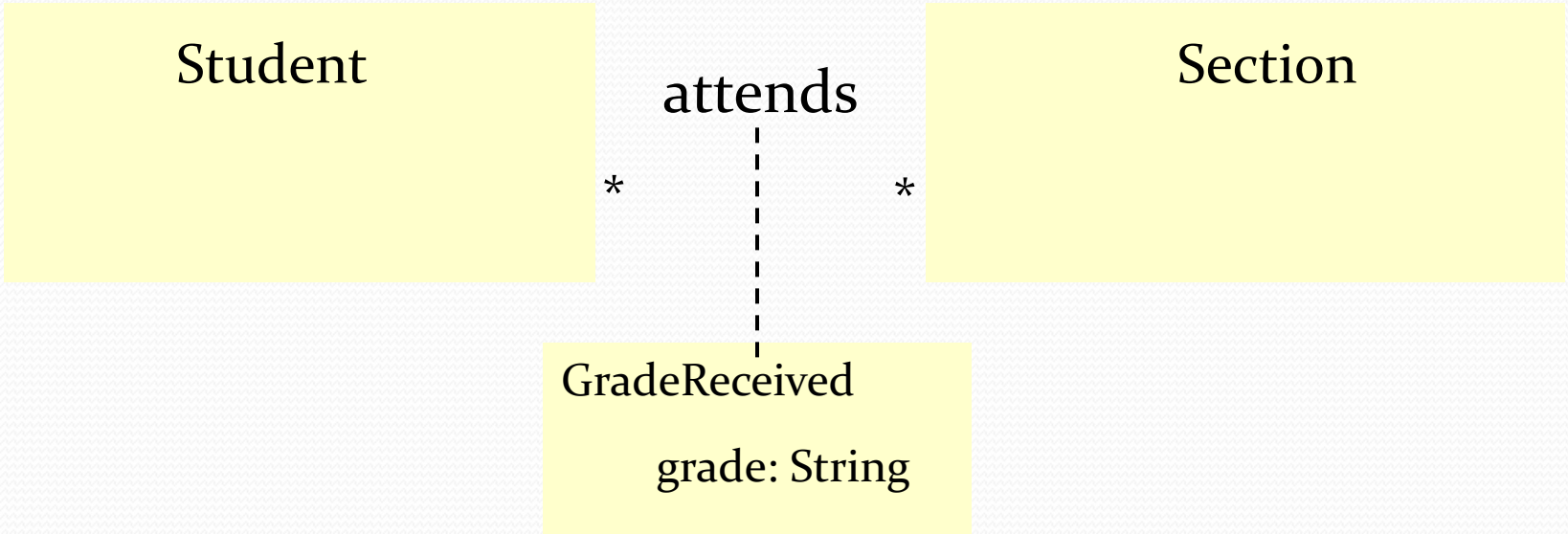
- Relationship between two or more objects of the same class.




```
public class Course {
    private List<Course> prerequisites;
}
```

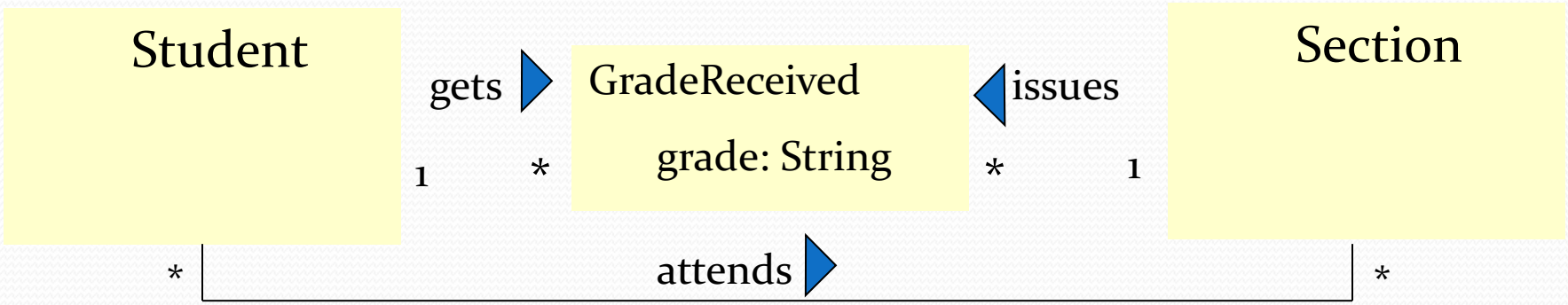
Association Classes

- We sometimes find ourselves in a situation where we identify an attribute that is critical to our model, but which doesn't seem to fit nicely into any one class.
- As an example, let's revisit the (many-to-many) association "a Student *attends* a Section,"
- Association Classes are useful to contain attributes of the link between objects.
- These are often modeled this way during analysis
- **Refer : Beginning Java Objects Pages file 425-429 or Book Page – 397-403**



Association Class - reworked

- During design, association classes are often re-worked to reflect the code instead of the concepts




```
public class Student {
List<Section> sectionsEnrolledIn;
List<GradeReceived> evaluationsReceived;
}
```

```
public class GradeReceived{
Student student;
Section section;
String grade;
}
```

```
public class Section {
int sectionNumber;
List<Student> enrolledStudents;
List<GradeReceived> gradeSheets;
}
```

Managing Bidirectional Many to Many Relationships

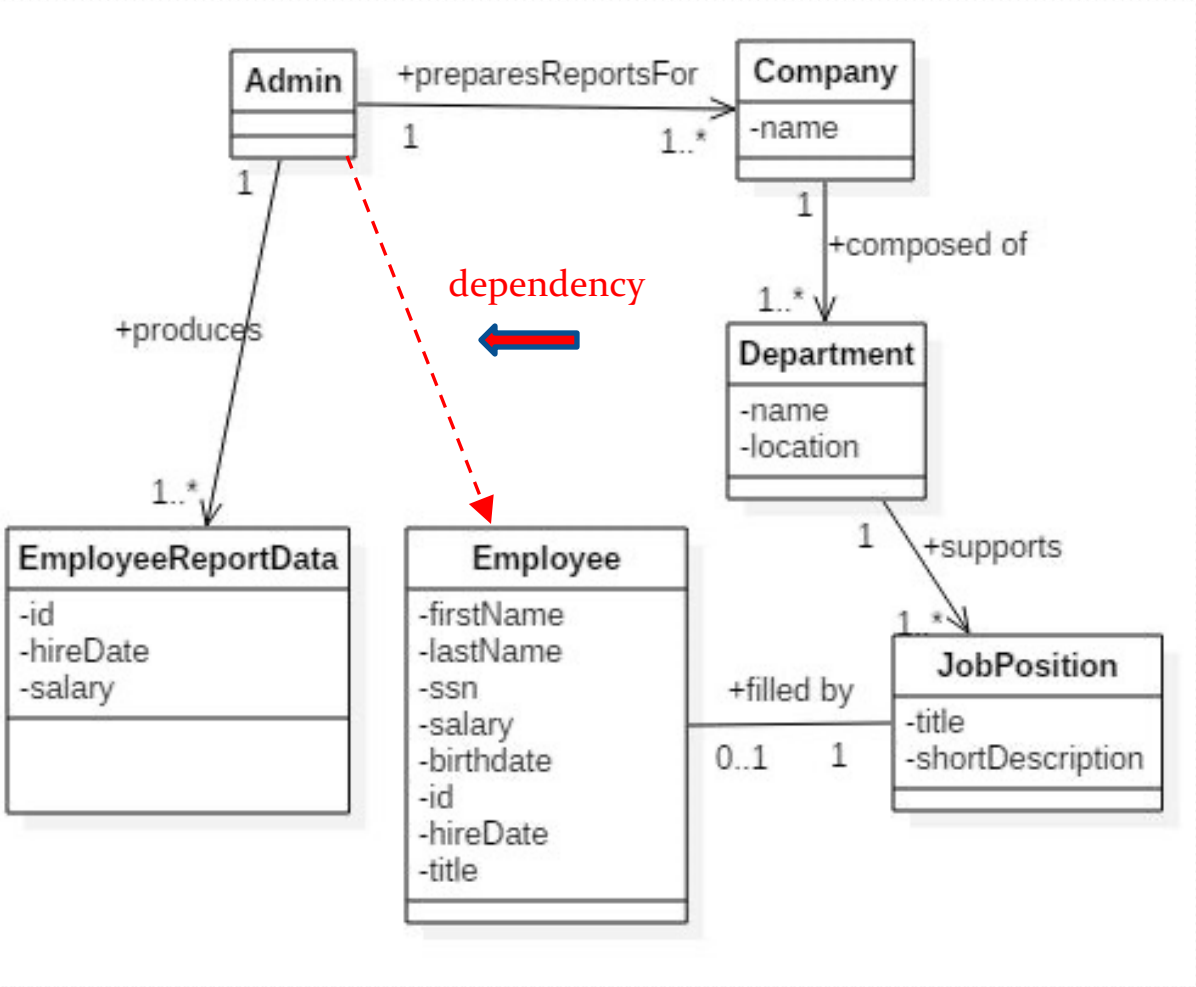
- This kind of implementation can be much involved and error-prone, but is sometimes necessary. Sometimes it is possible to refactor so that the association is only one-way (and therefore much easier to implement).
- The safest approach for implementing a many-many association is to use a factory class to create instances of the classes involved in the relationship.

Dependency

- An Association always implies a *permanent* relationship, since an instance of the target class is stored in the source class.
- Sometimes, only a *temporary* relationship is needed – for instance, an instance of a target class may be needed in order to read or set some values inside a method call, but the relationship need not endure after the method returns.
- Temporary relationships are modeled as *dependencies*.
- When creating a class diagram during analysis, assume all relationships are associations. Later (during design), review your work to see if some of the associations really ought to be dependencies.

Example

In the previous exercise, Admin does not need to keep a reference to all Employees in order to create EmployeeReportData – dependency is good enough



Main Point 3

There are several special forms of association, such as reflexive associations, aggregation, composition, and association classes, and dependency is a further refinement

Although most of these have their own symbols, you could still model these relationships without them.

The use of the symbols is to (easily) communicate additional information about the relationship.

Even these additional symbols are still based on the simple concept of an *arrow*. This is an example of diversity on the basis of unity.

Exercise 2.4

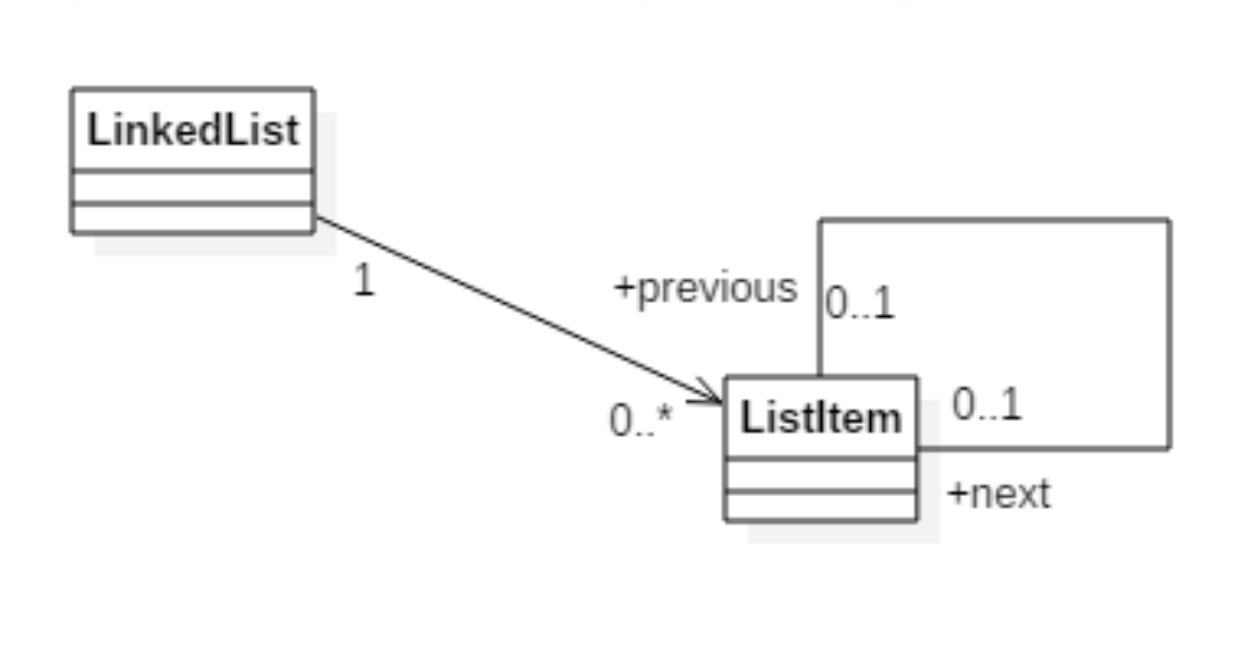
Draw a UML class diagram for the following.

Doubly-Linked List:

A LinkedList consists of zero or more ListItems. Each ListItem knows its previous and its next ListItem, if any.

Solution

Doubly Linked List



Summary

Modeling associations:

- We can use an association matrix to analyze what the relationships are between classes
- Associations are modeled with a line or arrow, and, optionally, a name describing the association, numbers on each side to indicate multiplicity, roles at either end of the association, and an arrow for directionality.
- Reflexive associations, aggregation, composition, association classes, and dependencies are further refinements of the concept of an association.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

1. Class diagrams are defined in terms of classes and their relationships (associations)

2. Although there are various special association forms (composition, aggregation, etc.), all are variations of the fundamental concept of an association from one object to another.

---

3. **Transcendental consciouness** is related to itself through its own self-referral dynamics.

4. **Wholeness moving within itself**: In Unity Consciousness, one recognizes that the relationship of the Self to the Self is not only fundamental, but is in reality the only relationship there is.