Lecture 3

1.4 Object-Oriented Paradigm

UML (Unified Modeling Language)

The Object-Oriented Paradigm

- *data* and *operations* are paired

- programs are composed of self-sufficient *modules* (*objects*)

  - each *module* contains all the information needed to manipulate its own data structure

  - and can interact with other *modules* (*objects*): send messages, receive messages, process data, ...

- Encapsulation, Inheritance, and Polymorphism

Objects and Classes

A *class* defines the abstract characteristics of a thing :

- its *attributes*
  (or fields or properties), and
- its *behaviors*
  (the things it can do, or *methods*, operations or features).

An *instance* of a *class* is called *object*.
Objects are represented using the same group of attributes, yet the values of those attributes may or may not be the same.

Attributes and *methods* of an *object* (*instance* of a *class*) are called its *members*.

Objects and Classes

Example: the obedient dog

Let's define a `class` `Dog`

`attributes`: `Name`, `BirthDate`, `Breed`, `HairColor`, `EyeColor`, `Weight`, `Gender`, `Location`

`methods`: `Sit`, `LieDown`, `RollOver`, `Fetch`, `Speak`

From this class we can create lots of instances of this class, that are called objects.

`Sunny`, `Spot`, `Lady`, …

To interact with them an entity *calls* one of its supported methods (called *invoking a method*);
this entity is called the *caller* and will often be some other object in the system.

Object-Oriented Philosophy

Encapsulation manages the complexity of a system. For programming this means:

- The *internals* of an object or class are invisible to the outside program
  (the *attributes* and the *code* for how methods work)
- The *attributes* are *private*
  (can only be accessed by code inside methods of the class)
- The *methods* are only visible by their *signature*
  (name, list of parameters and types, and the return type)

Object-Oriented Philosophy

Encapsulation manages the complexity of a system. For programming this means:

- The *internals* of an object or class are invisible to the outside program
  (the *attributes* and the *code* for how methods work)
- The *attributes* are *private*
  (can only be accessed by code inside methods of the class)
- The *methods* are only visible by their *signature*
  (name, list of parameters and types, and the return type)

Object-Oriented Philosophy

Encapsulation
is also called “information hiding” or “data abstraction”

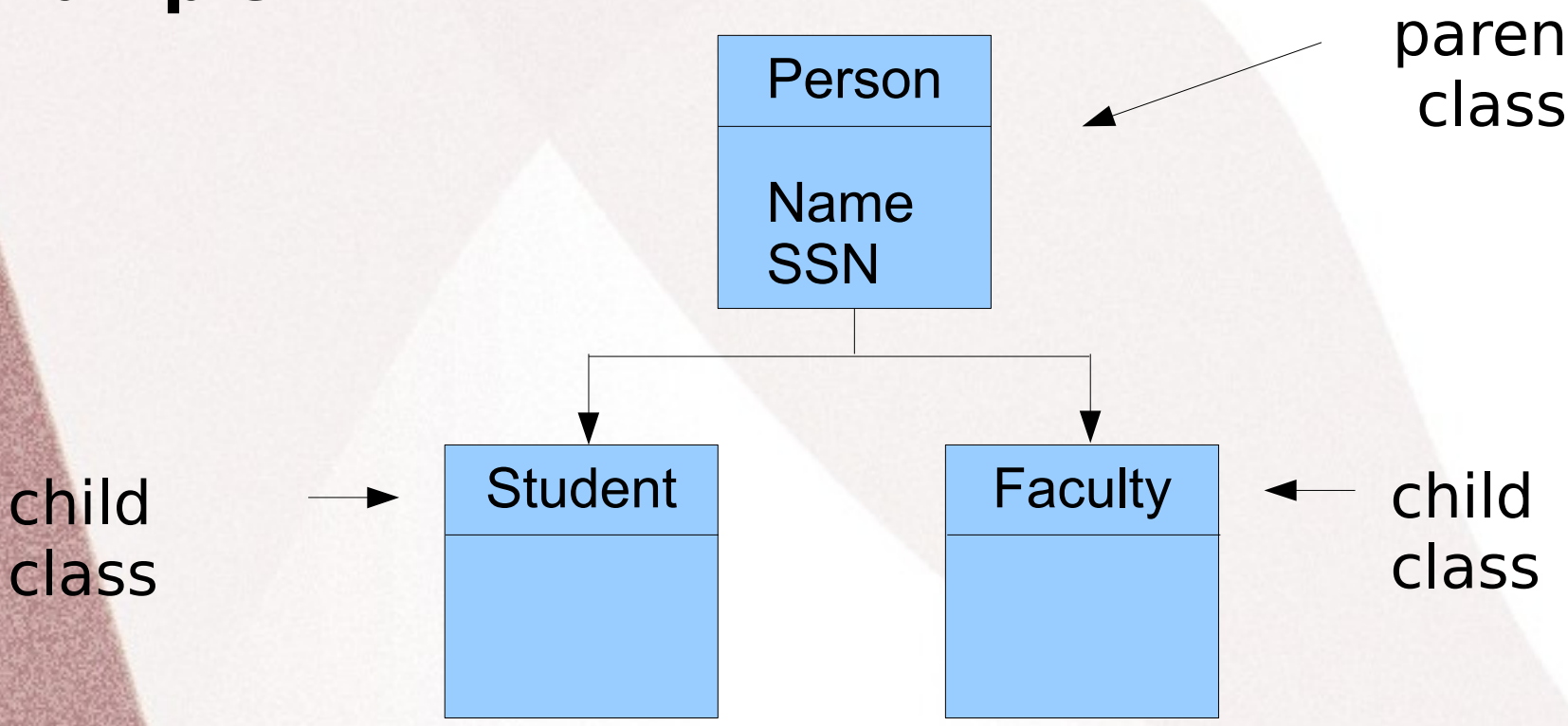
Example: a TV. Notice that we don't see what's inside it. How it works. It just works and we use it – choose channel, volume level and so on.

Object-Oriented Philosophy

Inheritance allows similar classes to re-use the same code for their methods, as well as their attributes.

this makes systems easier to maintain

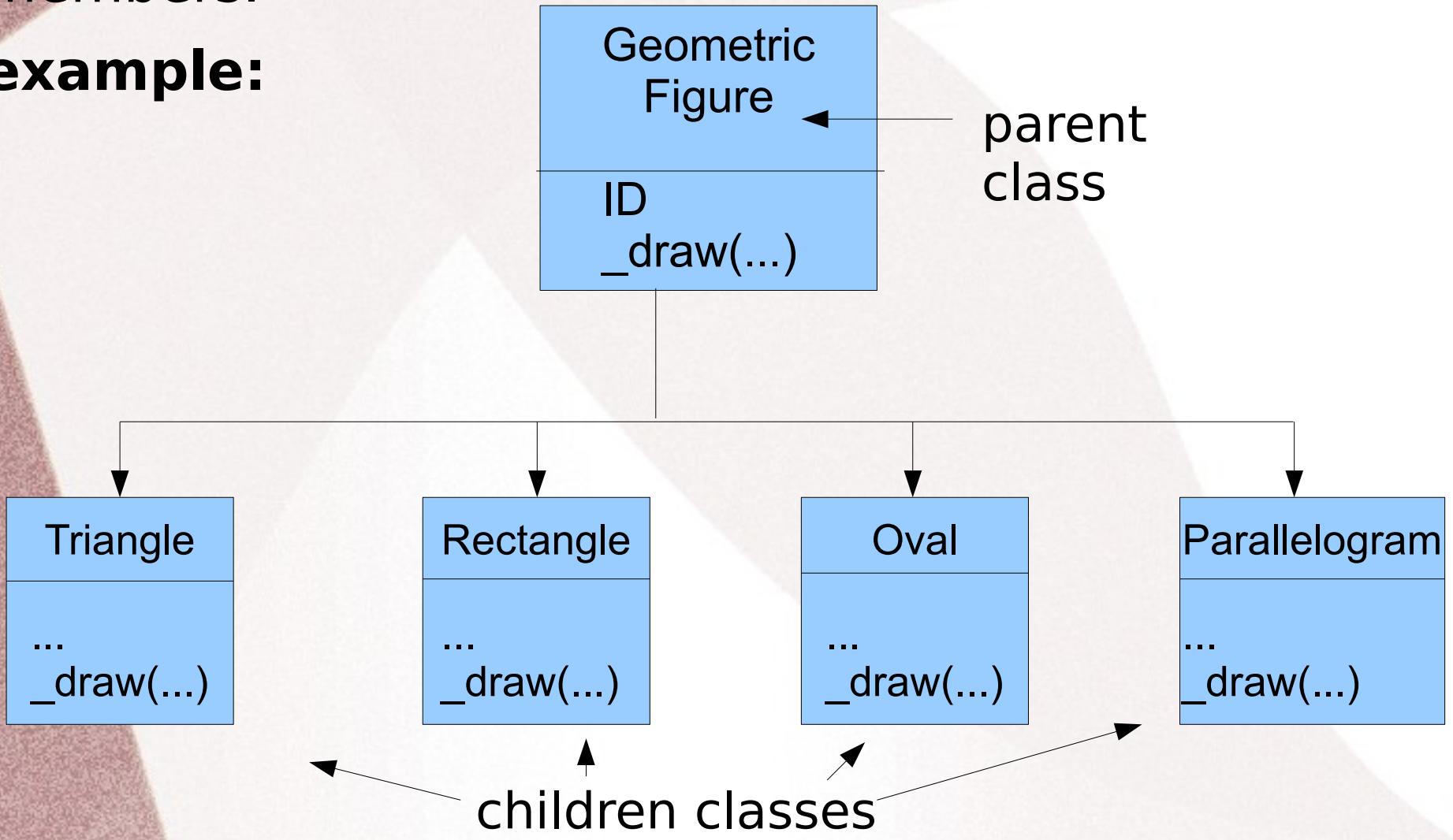
example:



Object-Oriented Philosophy

Polymorphism allows the programmer to treat derived class members just like their parent class's members.

example:



Object-Oriented Philosophy

For the time being let's postpone the talk about Object-Oriented Programming (OOP) and let's take a look at UML

Unified Modeling Language (UML)

UML is a standardized general-purpose modeling language in the field of software engineering.

The standard is managed, and was created by, the Object Management Group.

There are *four parts* to the UML 2.x specification:

1. the *Superstructure* that defines the notation and semantics for diagrams and their model elements (*current version: 2.2*);

2. the *Infrastructure* that defines the core metamodel on which the Superstructure is based (*current version: 2.2*);

3. the *Object Constraint Language* (*OCL*) for defining rules for model elements (*current version: 2.0*);

4. and the *UML Diagram Interchange* that defines how UML 2 diagram layouts are exchanged (*current version: 1.0*).

UML – when to use and why

One of the uses of UML is in the design of a software.

Is allows to visualize the software to be developed in multiple dimensions and helps to understand it.

A high-level model can be built at the beginning of the project, followed by more detailed ones during the process of development of the software.

After coding the ready software can be tested against a test model that is derived from the original model of requirements.

There is a very nice explanation of why and what for can we use UML:
http://www.cragsystems.co.uk/why_use_uml.htm

Modeling

There is a difference between the *UML model* and *the set of diagrams of a system*.

A *diagram* is a partial graphical representation of a system's model.

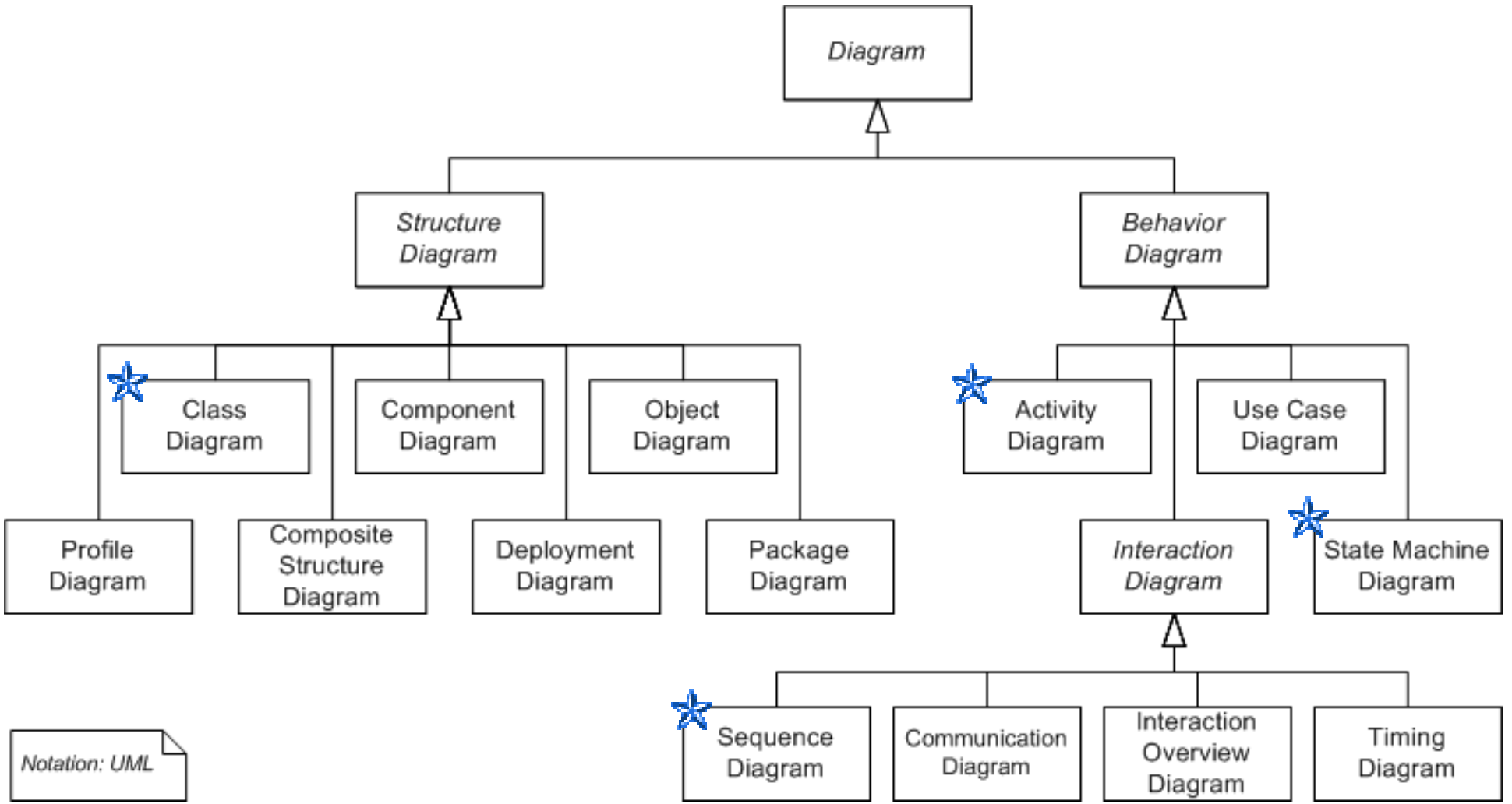
The *model* is a collection of diagrams.

UML diagrams represent two different views of a system mode:

- *Static* (or *structural*) *view*:
  Emphasizes the static structure of the system using objects, attributes, operations and relationships.

- *Dynamic* (or *behavioral*) *view*:
  Emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects.

UML diagrams



Source: Wikipedia
Author: Paulo Merson

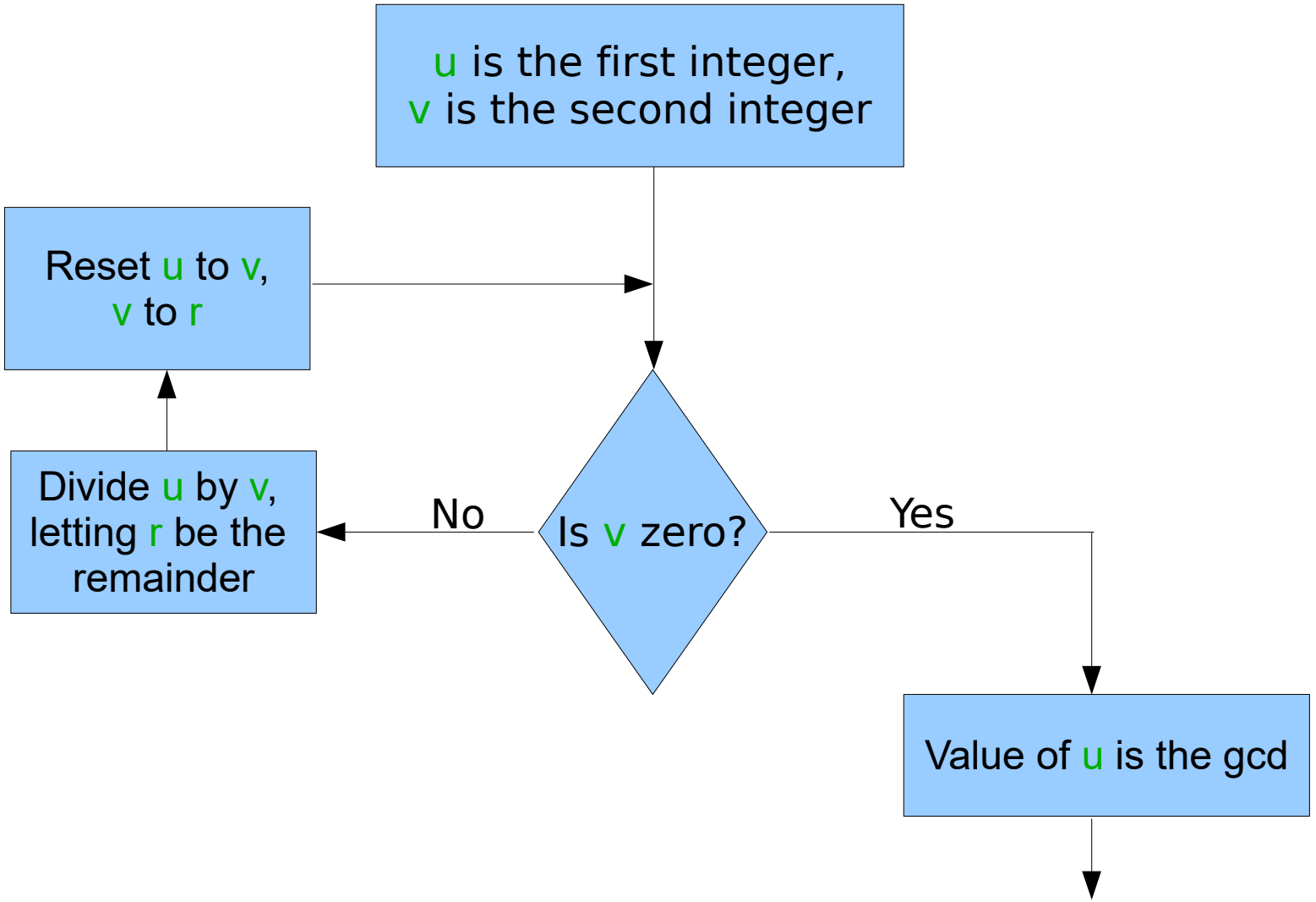
Let's take a look at few of the diagrams we will be using.

Activity diagram

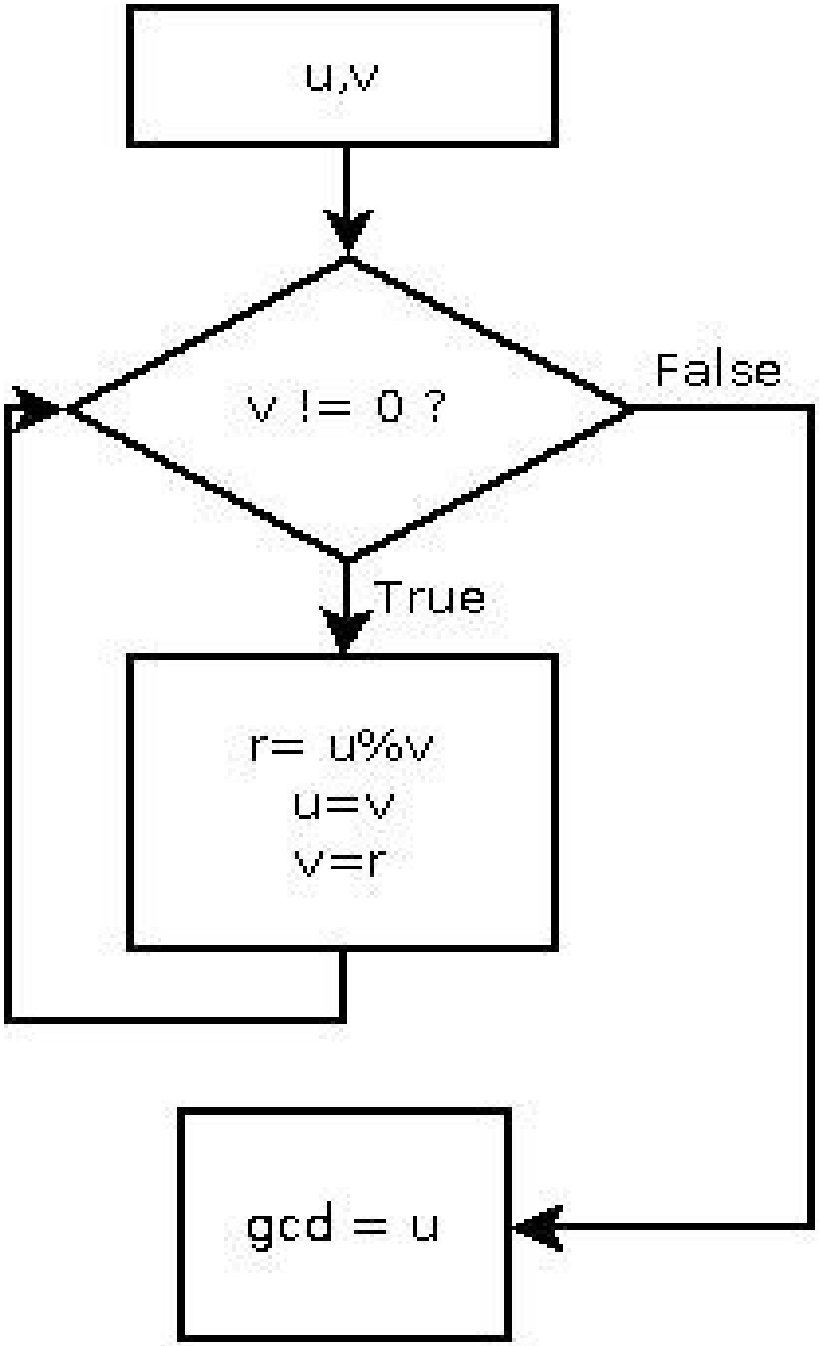
Activity diagrams are constructed from a limited set of building blocks consisting of nodes, activities, and decisions.

Although superficially similar to conventional flowcharts, activity diagrams also allow concurrent activities to be modeled.

Recall the Euclid's algorithm for finding GCD of two integers from Lecture 1. The flowchart we had:



Activity Diagram
(Flowchart)
For Euclid's algorithm
done in Dia 0.97:



Class diagram

A *class diagram* is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes.

In the class diagram these classes are represented with boxes which contain three parts:

- The upper part holds the name of the class
- The middle part contains the attributes of the class, and
- The bottom part gives the methods or operations the class can take or undertake

Let's recall the example with obedient dog.
We defined a `class Dog`

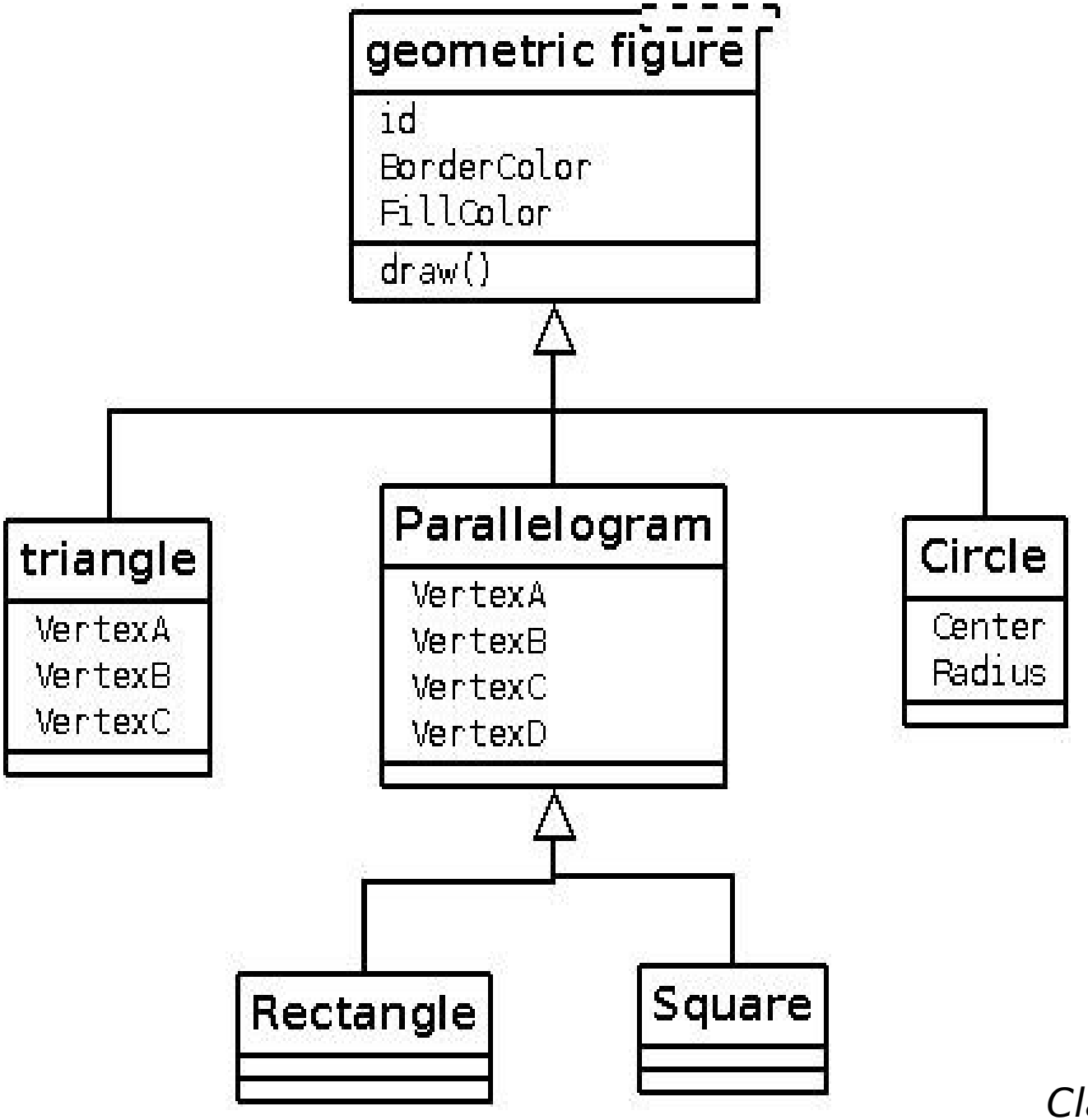
with

attributes: Name, BirthDate, Breed, HairColor, EyeColor, Weight, Gender, Location

and

methods: Sit, LieDown, RollOver, Fetch, Speak

Here is an example of a class hierarchy diagram:



Sequence diagram

A **sequence diagram** is a kind of interaction diagram that shows how processes/objects interact with one another and in what order.

- different processes (objects) live simultaneously as parallel **vertical lines** (*lifelines*) , and

- **horizontal arrows** show the *messages* exchanged between the processes in the order in which they occur.

dashed arrows – return messages 

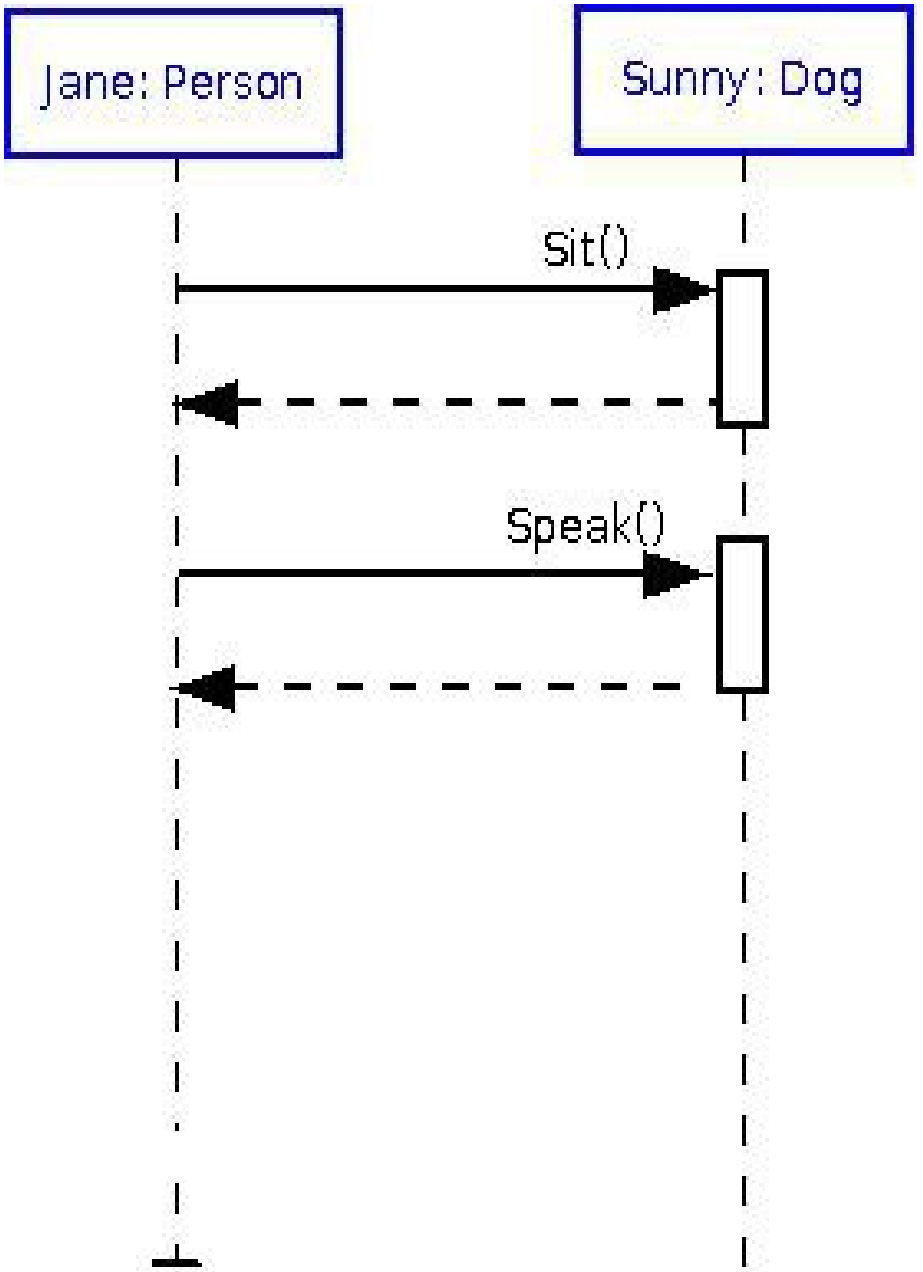
solid arrows - calls 

This allows the specification of simple runtime scenarios in a graphical manner.

We already have a class Dog and let Sunny be an instance (object) of that class.

Let's assume that we have a class Person, and Jane is an object of this class.

Let's see an example of interaction between a Jane and Sunny:



1.4.3 Designing a Student Registration system

Let's develop a Student Registration System.

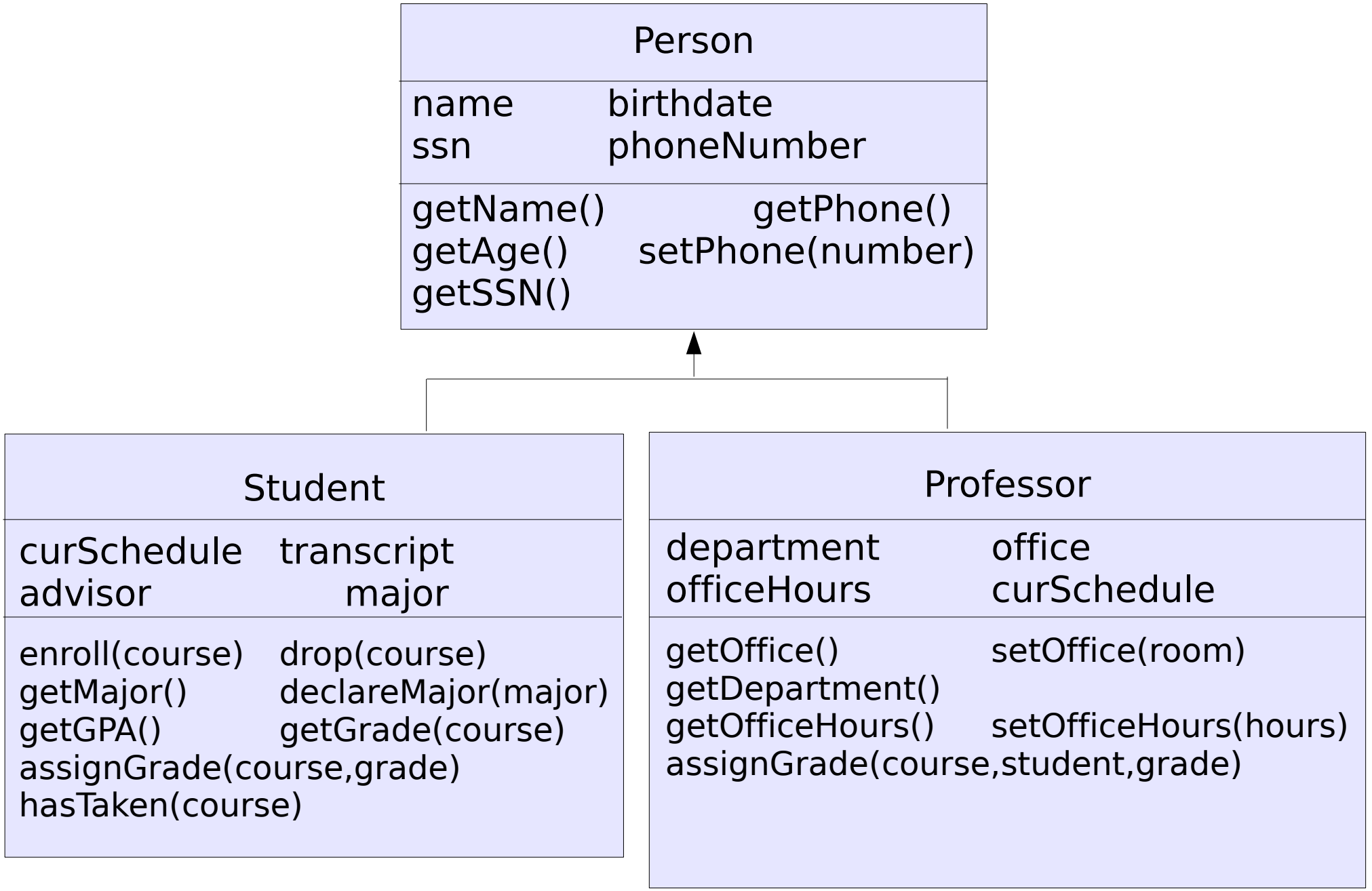
Before any implementation/coding starts its good to realize what do we want from the future system and to show the structure schematically (using diagrams):

- what do we want from the system:
Student should be able to register for a course, and drop it. Each course has a list of students, time, place and Professor's name. Professor has a list of courses to teach, belongs to a Department. Students have majors. Sometimes students have to meet some pre-requisites of the course to be able to take it.

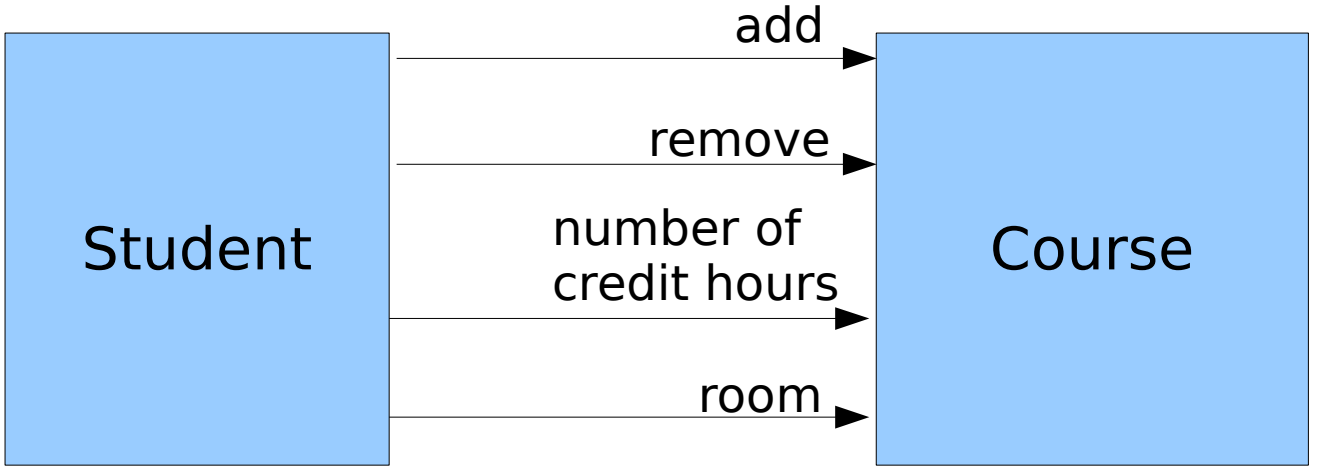
- To show the structure, we will define classes: Student, Professor, Course, ...

Class Diagrams

CSI 32



- let's take a look at possible relationships between *Classes*.



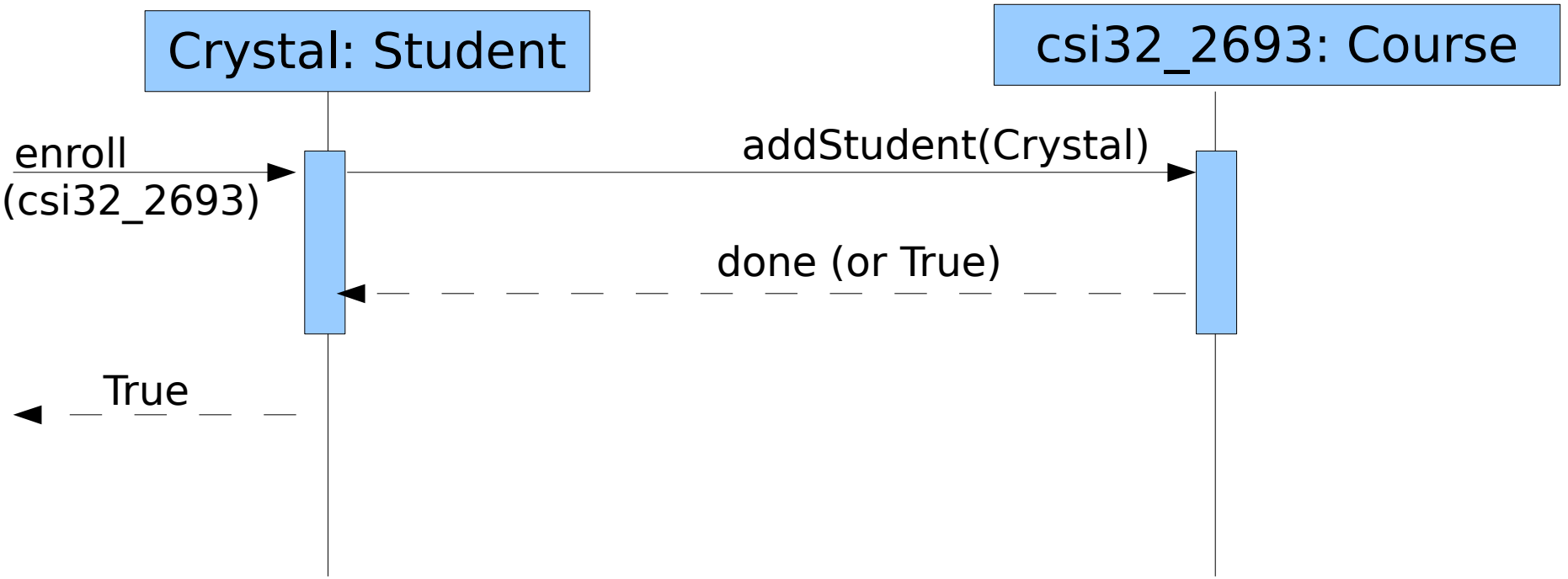
. . .

Class Diagrams

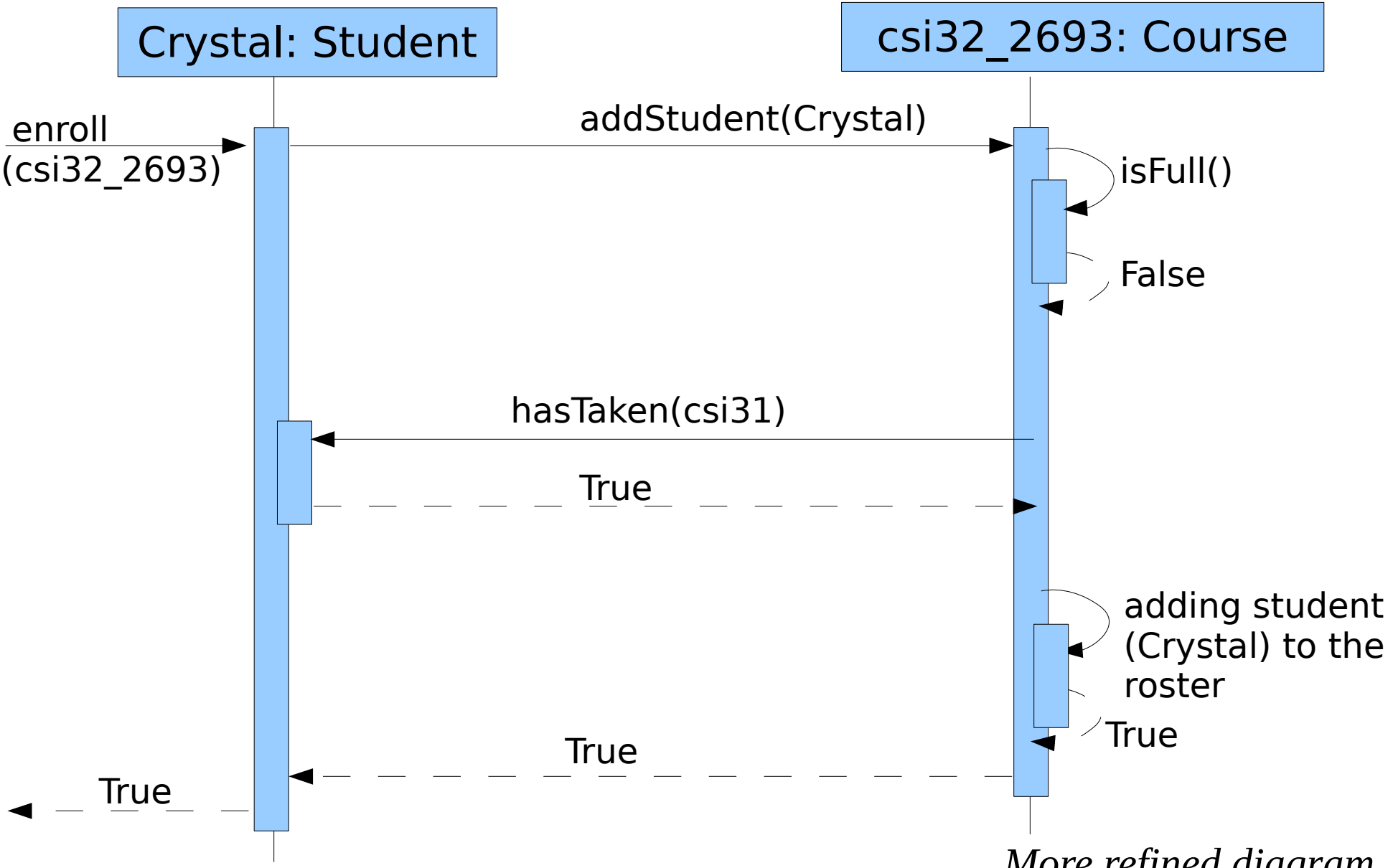
Course
title prerequisite classLimit room number instructor listOfStudents hours
getListOfStudents() addStudent(student) removeStudent(student) getClassLimit() getNumOpenSeats() isFull() getInstructor() setInstructor(professor) getTitle() getNumHours() GetRoom() getDepartment()

Department
title listOfInstructors listOfCoursesOffered listOfStuff room   hours
getTitle() getListOfInstructors() addInstructor(professor) removeInstructor(professor) getListOfStuff() addStuff(person) removeStuff(person) getRoom() getListOfCoursesOffered() addCourse(course) removeCourse(course)

Sequence Diagrams

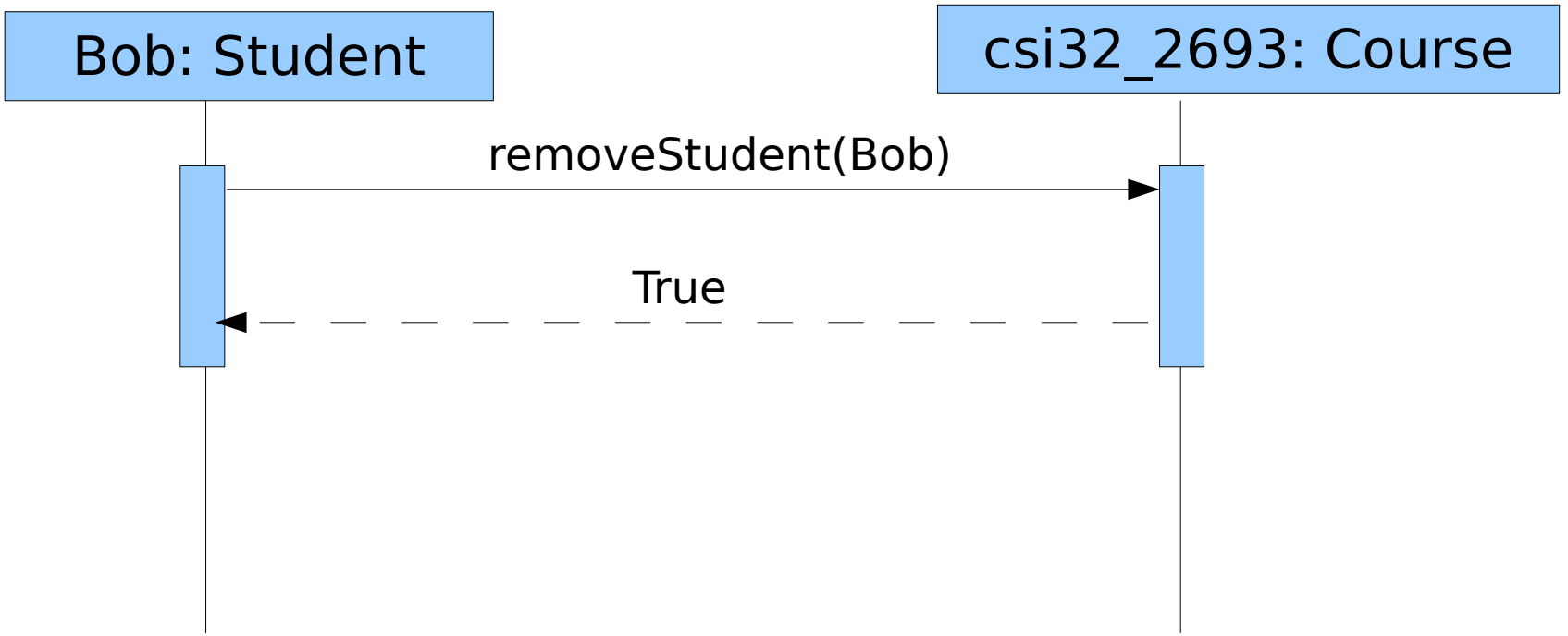


Sequence Diagrams



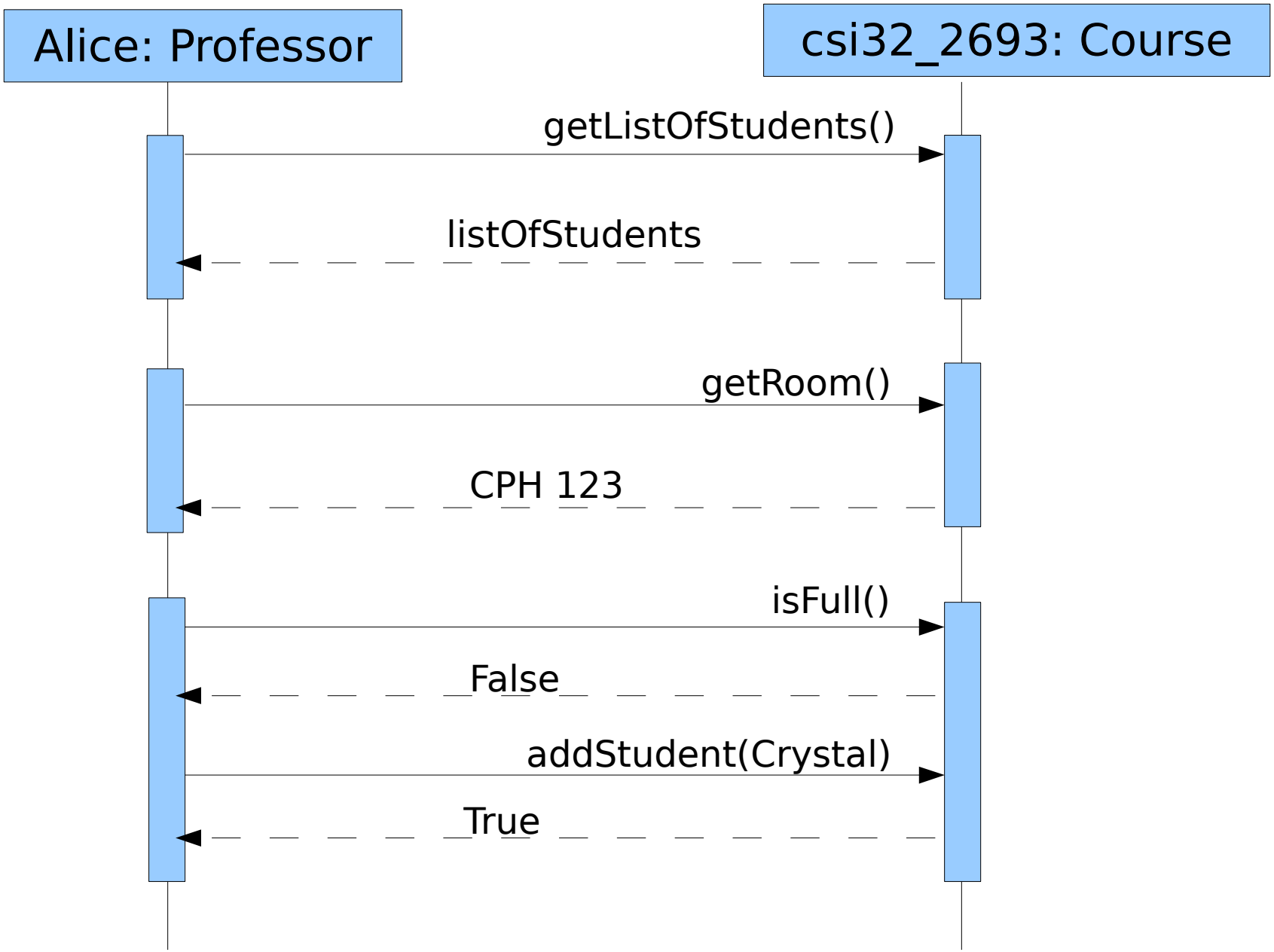
More refined diagram

Sequence Diagrams



Bob decided to drop a class

Sequence Diagrams



In-class work

1. Build an *activity diagram* for the following algorithm:

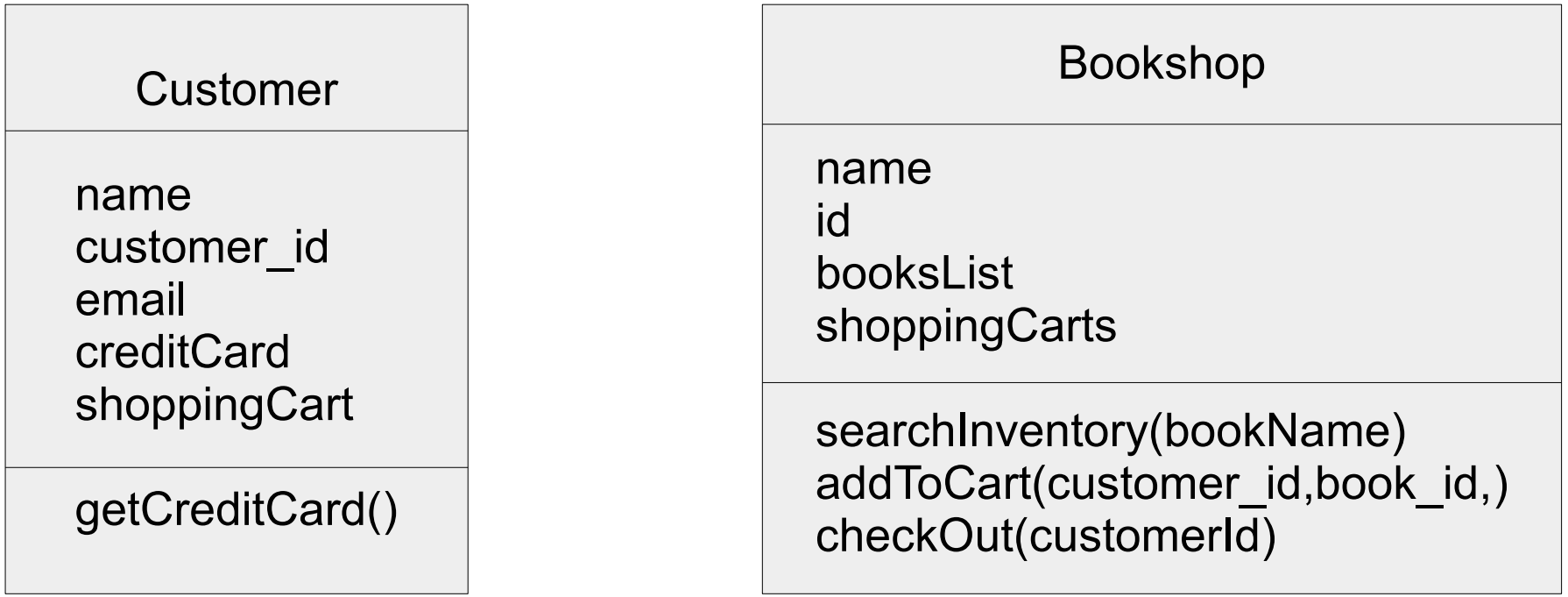
```
a=10
b=20

while a > 0:
   b+= a*b
   a -= 2

print(a,b)
```

In-class work

2. Design and draw a *class diagram* for a class for an **online bookstore**

In-class work

3. The class diagrams of an online customer and an online bookstore are given below.
Tom (with **id=11768**) wants to check if the online bookstore named **shop23** has the book "*Landscaping Design*" in stock (assume that it is in stock and has **id=123**) and to buy it if it is in stock. Drawn a sequence diagram modeling the purchase.



Homework Assignment

Page 29 / 1.19
Page 30 / 1.20, 1.22, 1.27, 1.31

Comments:

For 1.20: The book has an answer for 1.20, but try to do it on your own first, and then, if needed, refer to the solution.
Please, note that we did not create Schedule class and hence didn't use it in our scenarios, so you shouldn't use it either.