

Lecture 2

1.4 Object-Oriented Paradigm

UML (Unified Modeling Language)

The Object-Oriented Paradigm

- *data* and *operations* are paired
- programs 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*. Object 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*.

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*), and 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)

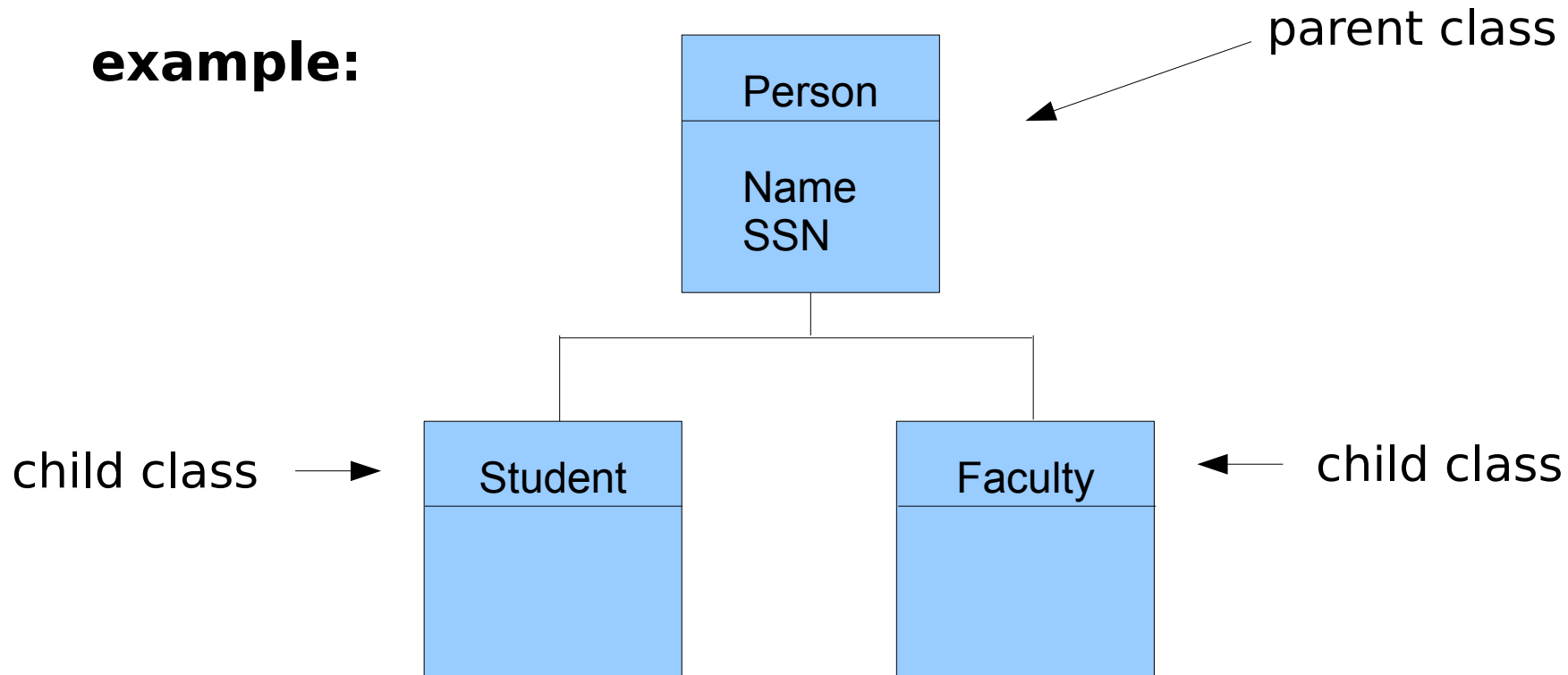
Encapsulation is also called “information hiding” or “data abstraction”

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

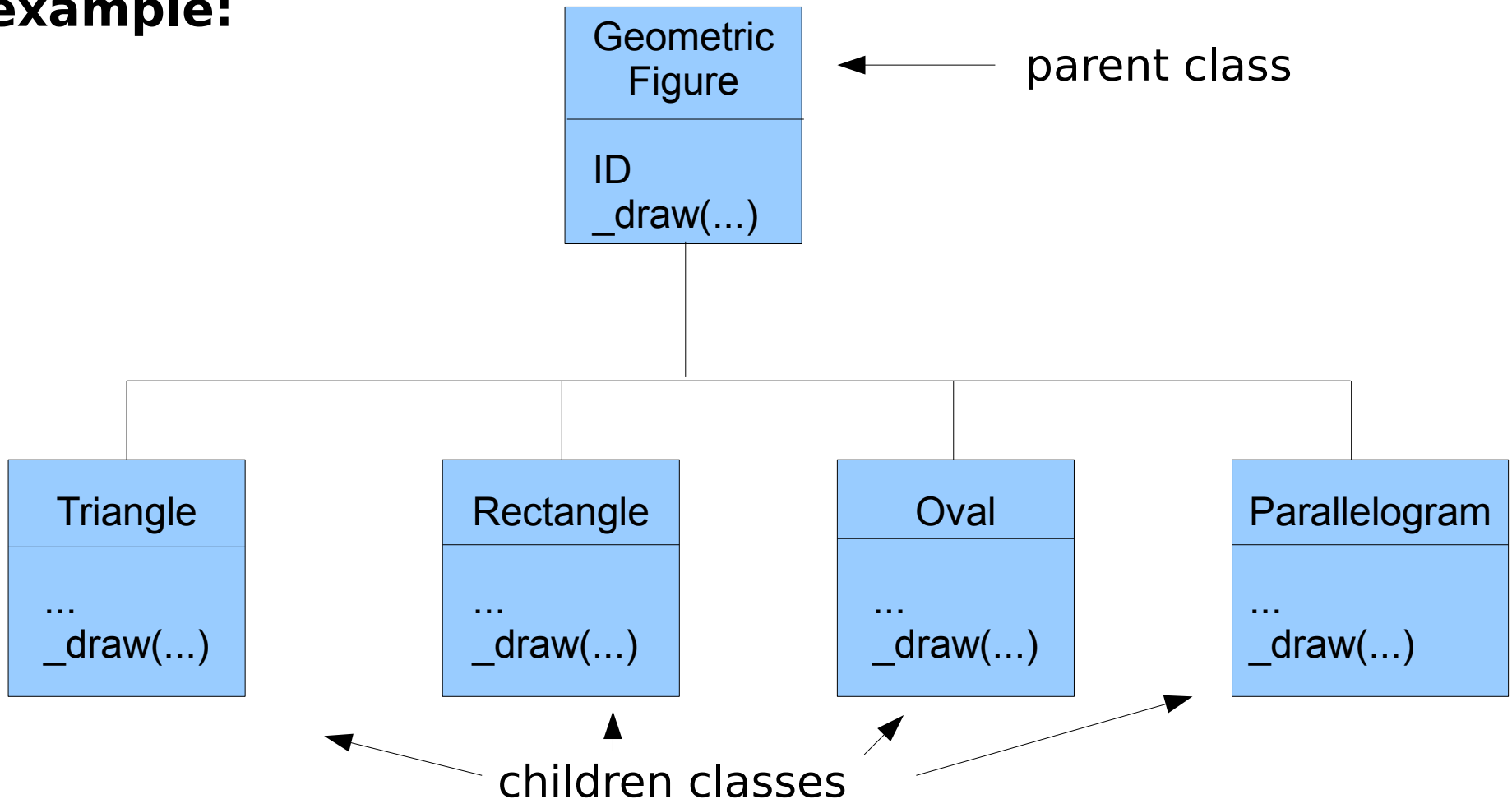
example:



Object-Oriented Philosophy

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

example:



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.

It 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.cragssystem.co.uk/why_use_uml.htm

Modeling

It is important to distinguish between the *UML model* and *the set of diagrams of a system*.

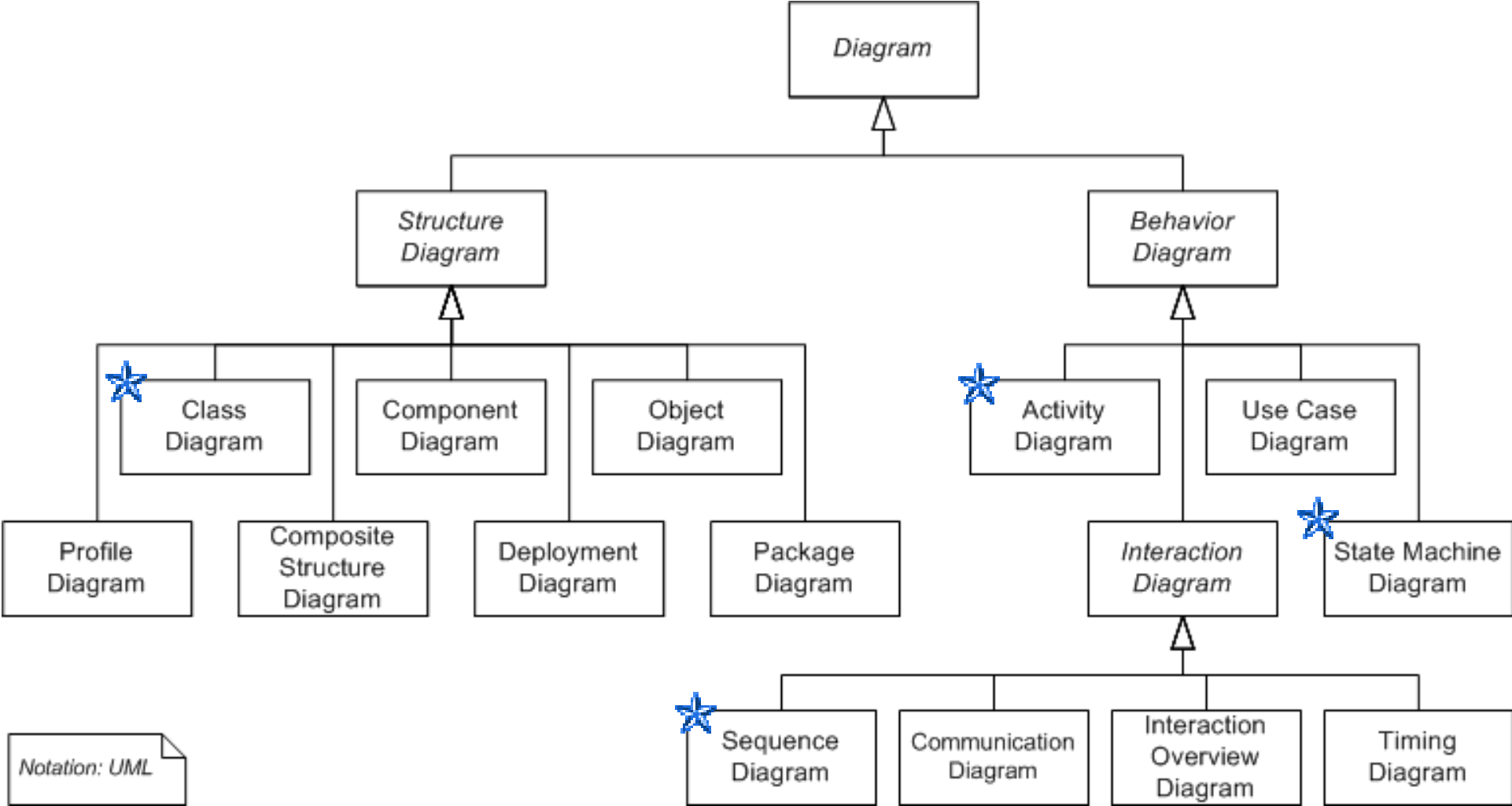
A *diagram* is a partial graphical representation of a system's model.

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



Notation: UML

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

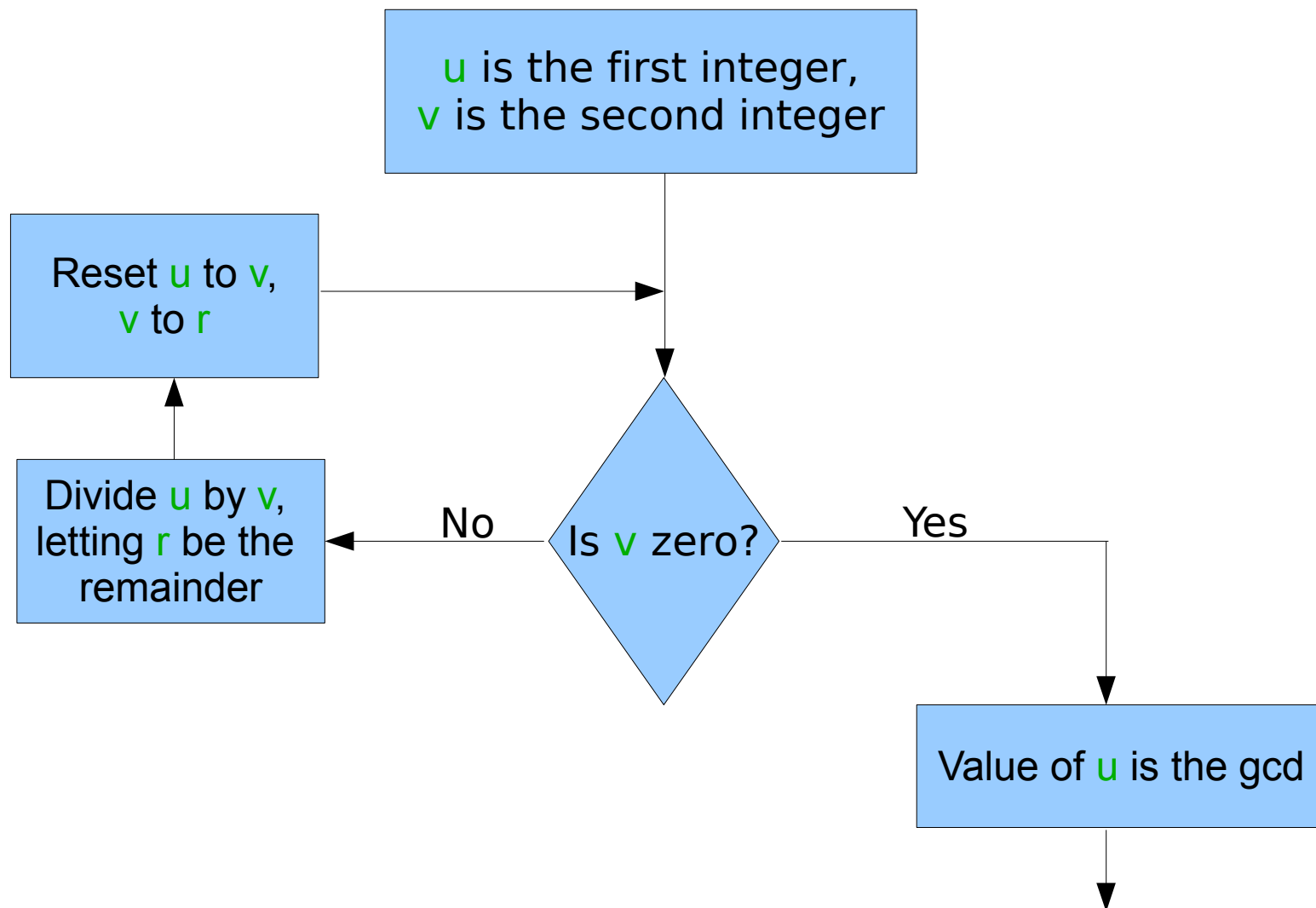
Activity diagrams

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

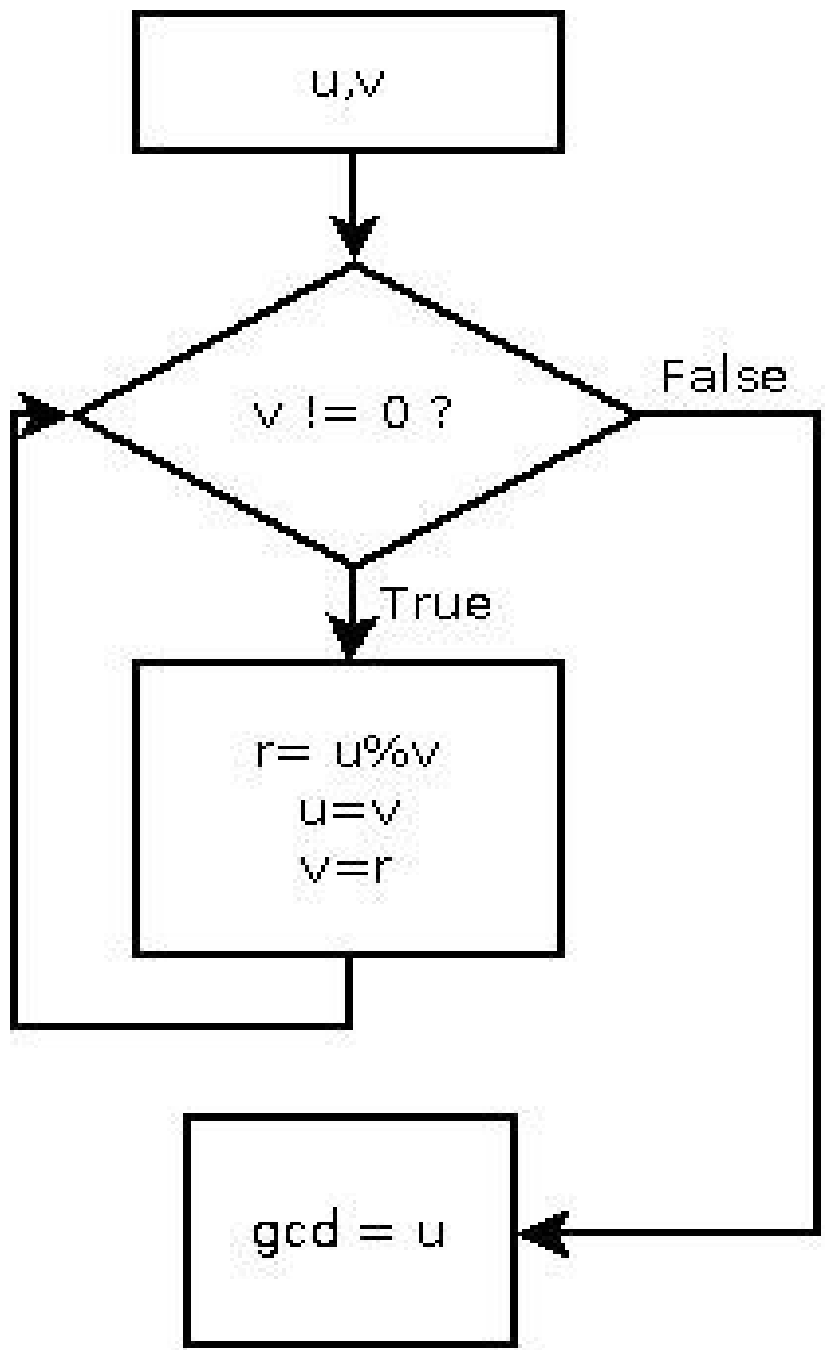
Although superficially similar to conventional flowcharts, activity diagrams also allow concurrent activities to be modeled.

CSI 32

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`

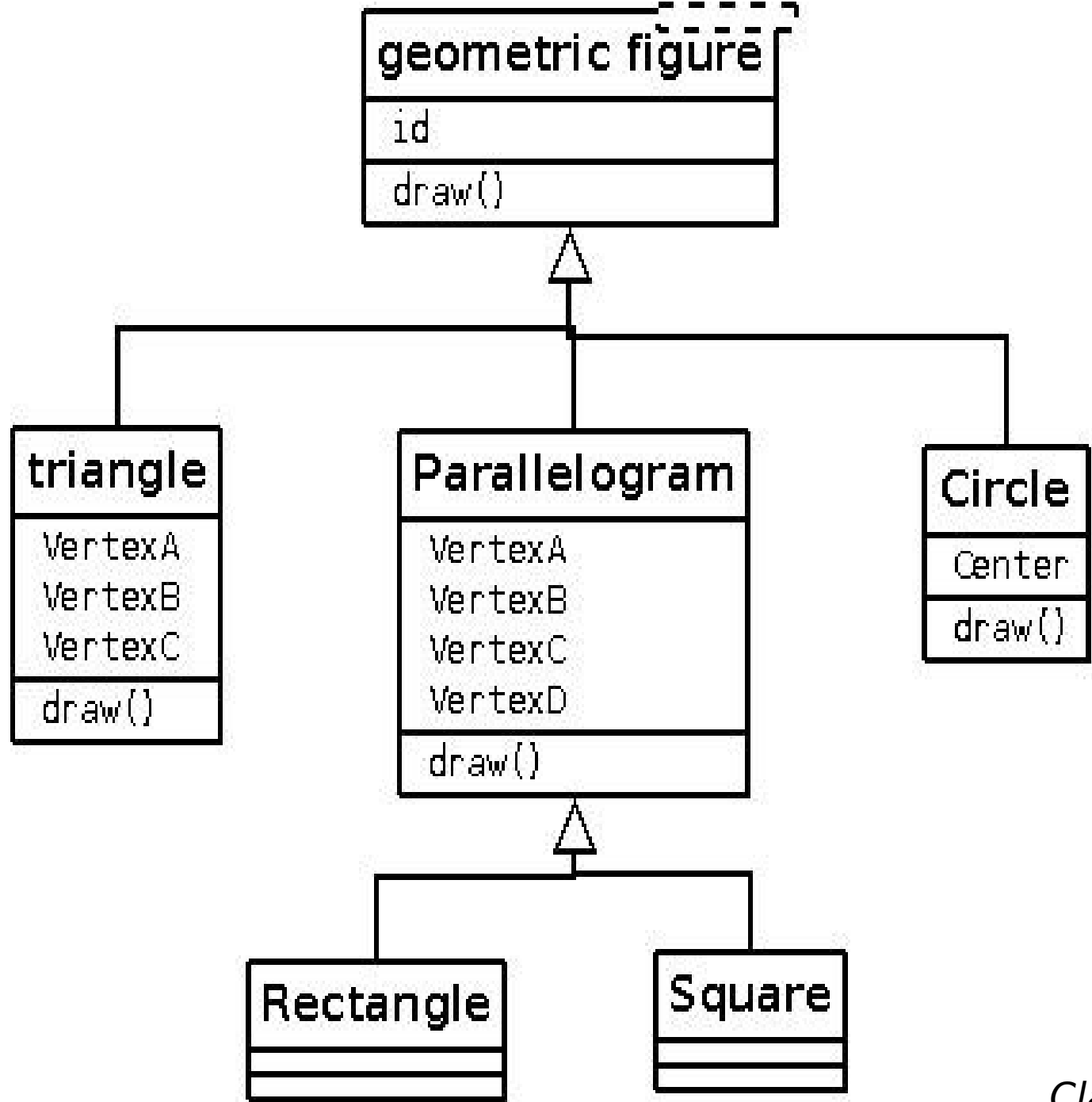
and

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

To the right is the class diagram for this class



Here is another example of a class diagram:



Sequence diagram

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

It shows different processes or objects that live simultaneously as parallel **vertical lines** (*lifelines*)

and,

as **horizontal arrows**, the *messages* exchanged between them, 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**:

