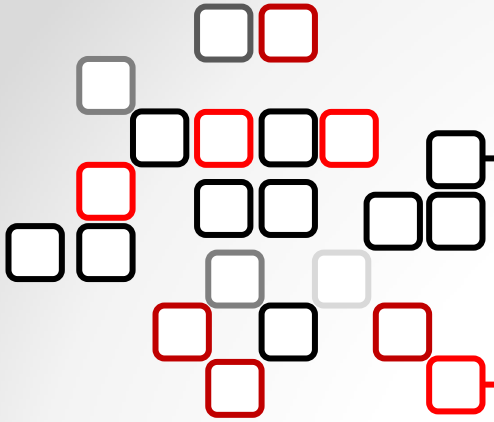
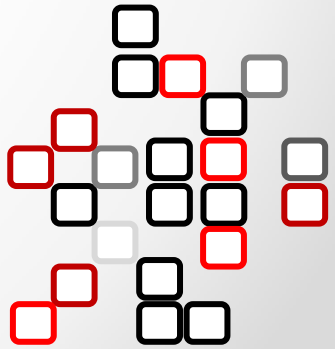
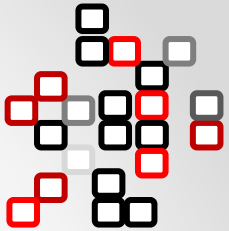


Autonomous Systems

UML LECTURE - REPETITION
Object-oriented modelling
Introduction into the UML



Content



- Introduction into the object-oriented modelling
  - Comparison with procedural approach
  - Object vs. Class
  - Hierarchy
- Object-oriented principles
  - Abstraction
  - Inheritance
  - Encapsulation
  - Message sending
  - Polymorphism
- UML

Structural approach

- **Structural (procedural) approach**
  - **A computer program is perceived as a collection of programming blocks (procedures) that can run from the main program**

```
procedure sumValues (a, b) {
 int c = a + b;
 print "Sum of values is:" + c;
}

function giveMeResult (c, d){
 ...
 return result;
}

begin
        sumValues();
        int d = giveMeResult (c, d);
        ...
end
```

Main program

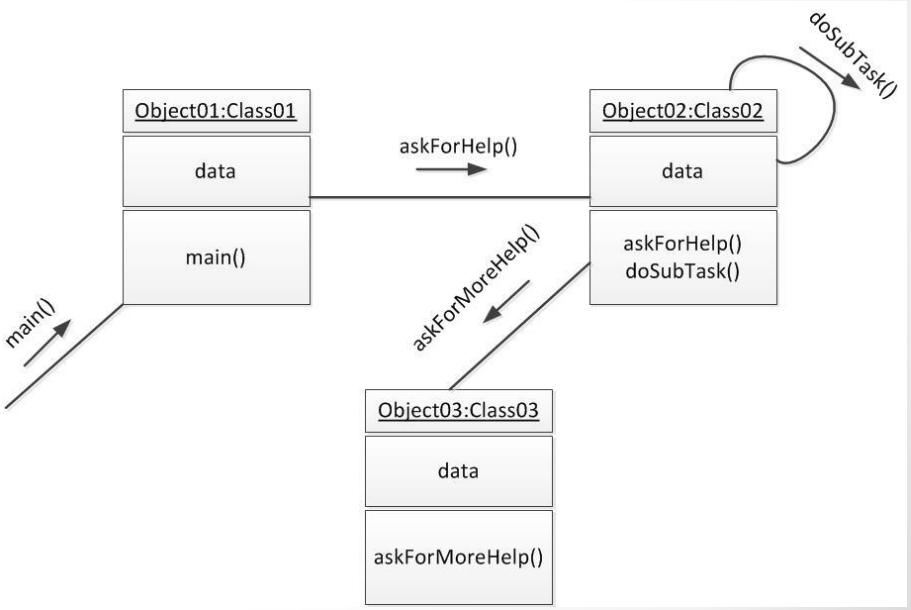
Complexity of the system grew and programmers have to divide development of the system into smaller programmes that would be easily maintanable.

Disadvantages of the approach
-difficulties in finding errors and repairment of these errors
- repetition of blocks with programming codes in various places of the whole program

Object-oriented approach

- **Object-oriented approach**
  - **Object-oriented program is perceived as a collection of compact units – objects able to interact (communicate) with each other**

Advantages of the OO approach
-potential for reduction of errors and costs
- higher flexibility of OO-based systems
- each object is relativelly small (self-contained) and easily maintainable
- complexity reduction (objects as compact units)
- objects can be reused in different systems

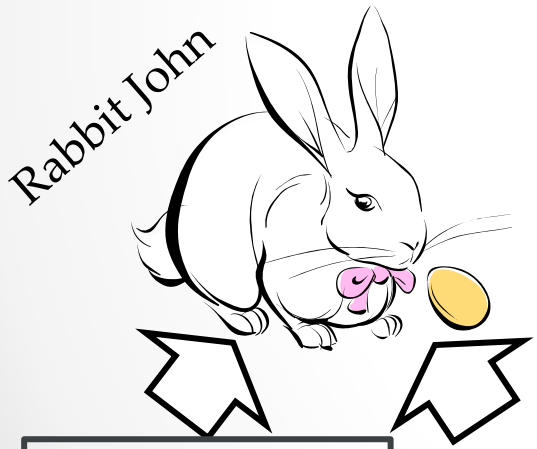


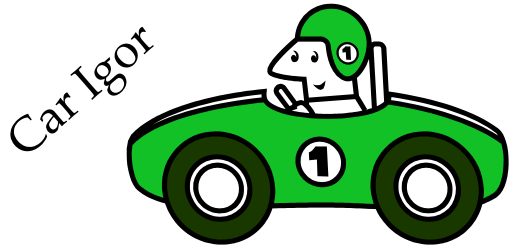
Object

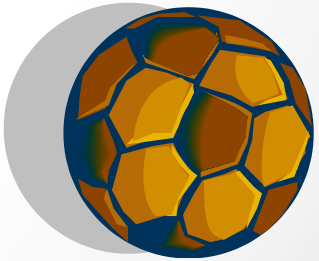
- **Object**
  - **Self-contained unit having concrete attributes and behaviour (operations, methods)**
  - **Something what can be explicitly defined**
  - **Mentally bordered part of the reality that exists in time and what consists of a group of states, relations and behaviour**
  - **Something what has the group of operations (methods) and state that is able to remember effects of operations**
  - **Object has a state, behaviour and an identity**

Object

- **Objects know about their state and have own independent decision making mechanism (own inner logics) that determines behaviour of object**
- **Each object is unique**







colour: white
age: 2
size: small

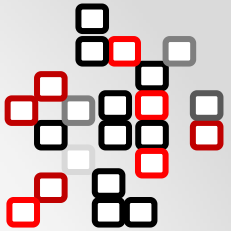
eat
sleep
relax

colour: green
tradeMark: Škoda

go
break

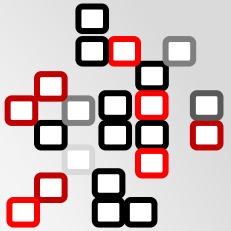
State | **Knows things** | **Knows how to do things**

Characteristics of objects



- Characteristic (property) is an abstraction of the particular state
  - characteristic (property): age <- abstraction
  - particular state: 21 years
- Property defines the identity of the object and its uniqueness
- Specification of a property is the attribute (design concept)
- Attributes of objects = NOUNS
- Small letter without gaps: age, streetName, …

Behaviour of objects

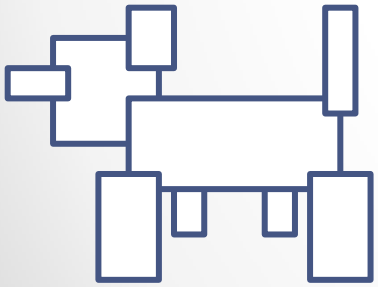


- **Behaviour of an object:**
  - **What the object do or can do.**
  - **How the object move from one state into one another.**
  - **How the object change its attributes in time.**
- **Operation (analytical concept): what the object do or can do**
- **Method (design concept): how the object will do something**
- **Methods of objects = VERBS**
- **Imperative shape: make(), studyBook(), relax(), calculateEquation(), …**
- **Small letter without gaps: above example …**

Class

- „Type of thing“
- It plays the role of the template that is used for objects building (instantiation)
- Object is an instance of the class (i. e. particular realisation of the class)

Template Class Object Instance of class

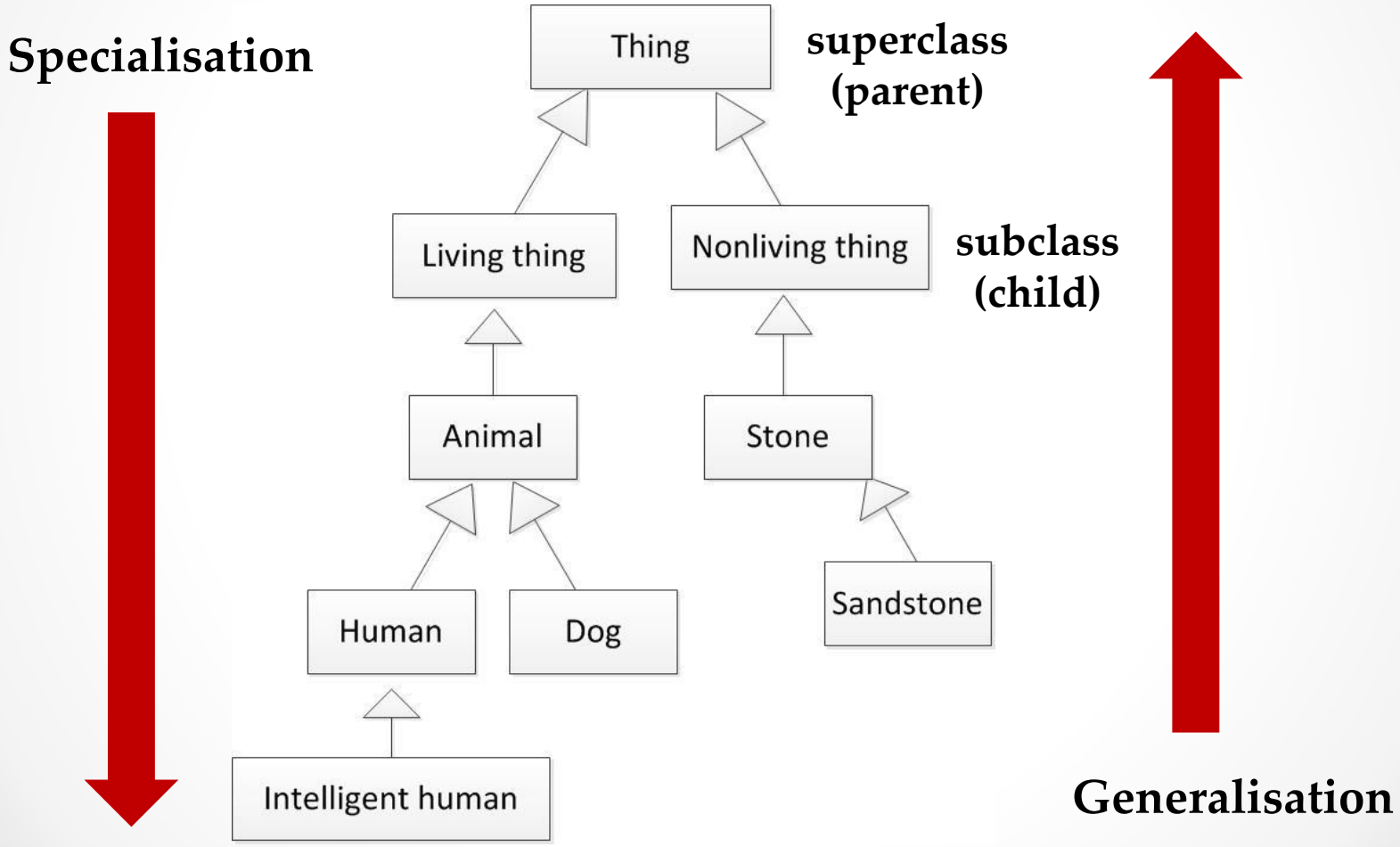


Dog
breed
age
lengthOfHair
colourOfHair

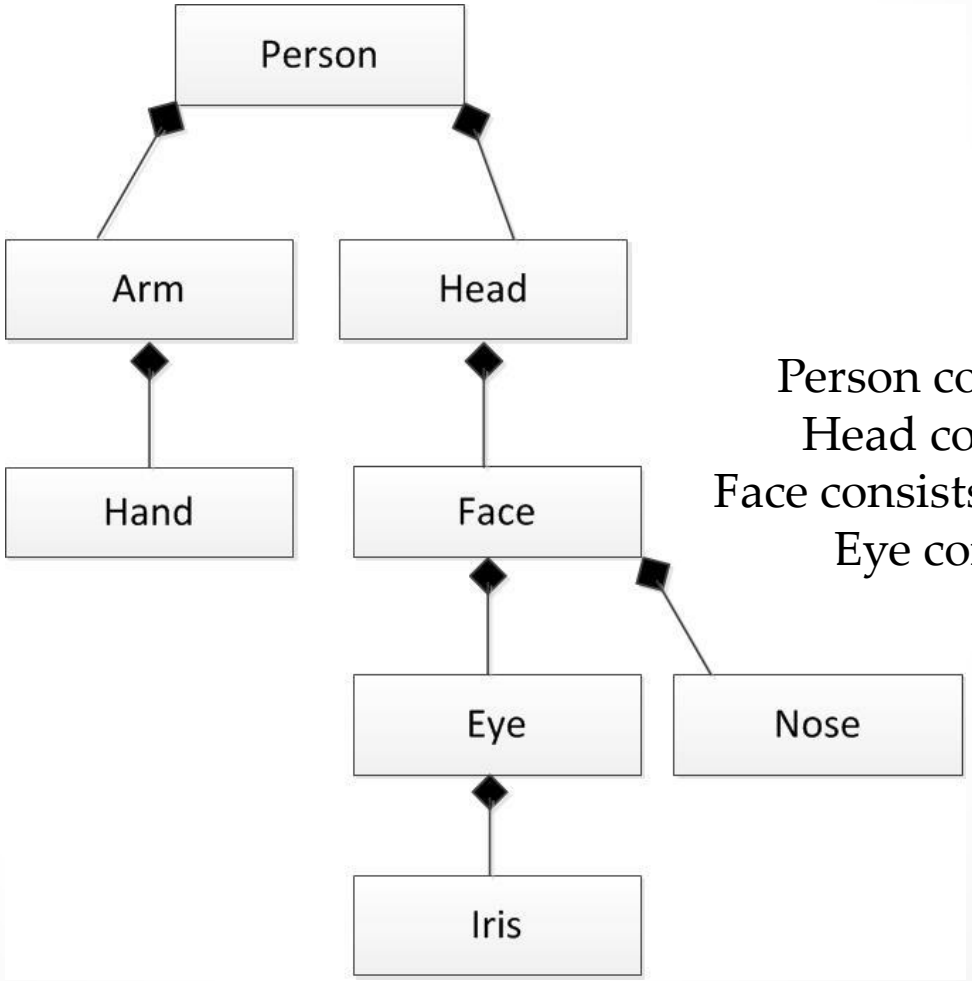
WinstonChurchill
breed: PetitBrabancon
age: 1
lengthOfHair: short
colourOfHair: brown

Capital letter without gaps: Dog, MusicalSchool, …

Hierarchy of classes (TYPE 1) Generalisation/Specialisation

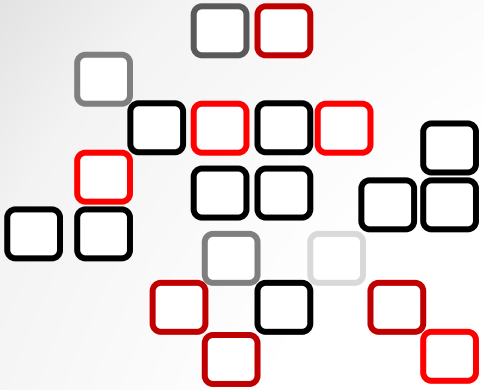


Hierarchy of classes (TYPE 2) Part-Whole relationship



Person consists of Arm.
Arm consist of Hand.

Person consists of Head.
Head consists of Face.
Face consists of Eye and Nose.
Eye consists of Iris.



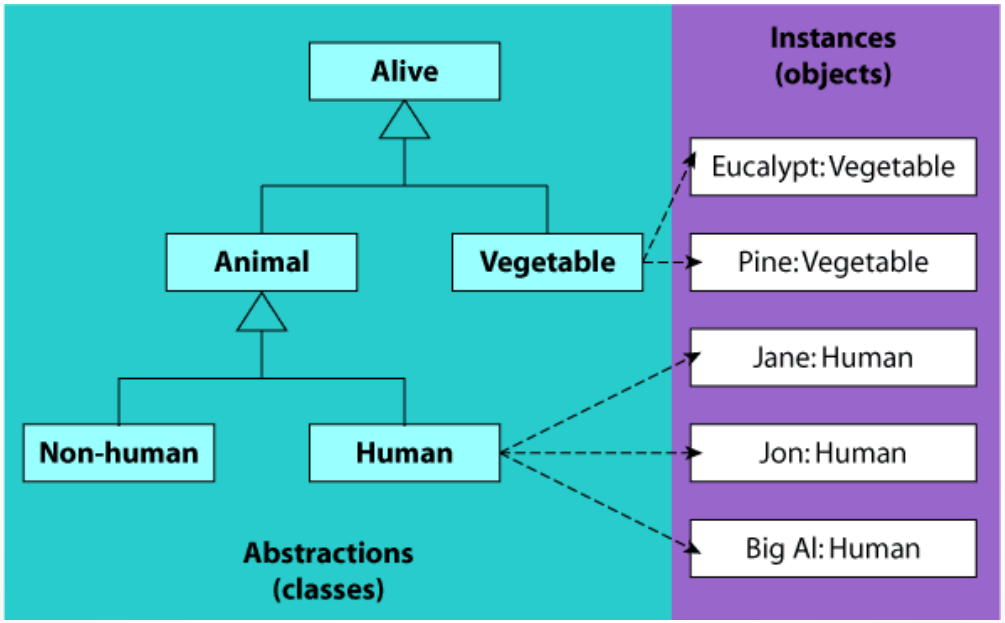
We will go much deeper

object-oriented principles

. . .

Principle 1: Abstraction

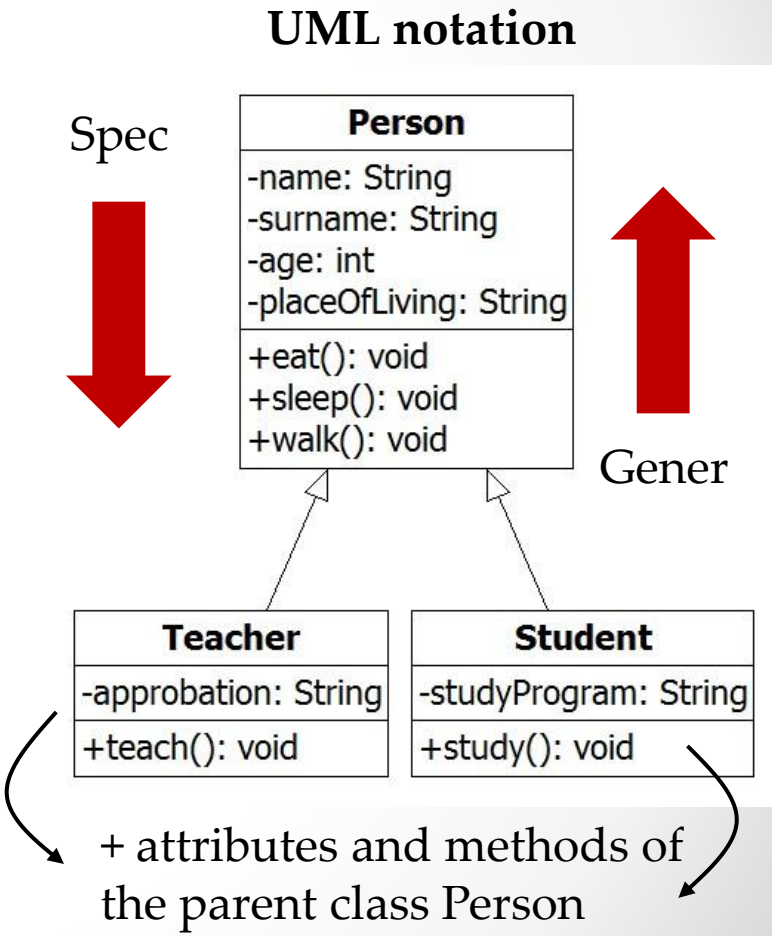
- Simplification of the complexity of the application area that „will be integrated“ into the software system
- It is the „art“ to focus only on the substantial properties and behaviour that are necessary to be included into your software



http://www.csse.monash.edu.au/~jonmc/CSE2305/Topics/01.01.OverviewAbsEnc/html/text.html#abstraction_or_#34all_men_are_animals#34

Principle 2: Inheritance (1)

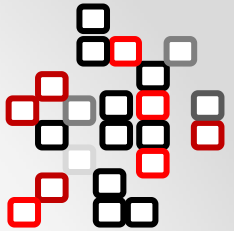
- **Hierarchy of the 1st type (generalisation/specialisation) specifies the inheritance**
- **Relation where one class share attributes and behaviour declared in a different class (parent) and has own attributes/behaviour that are located only in this class (child)**



Principle 2: Inheritance (2)

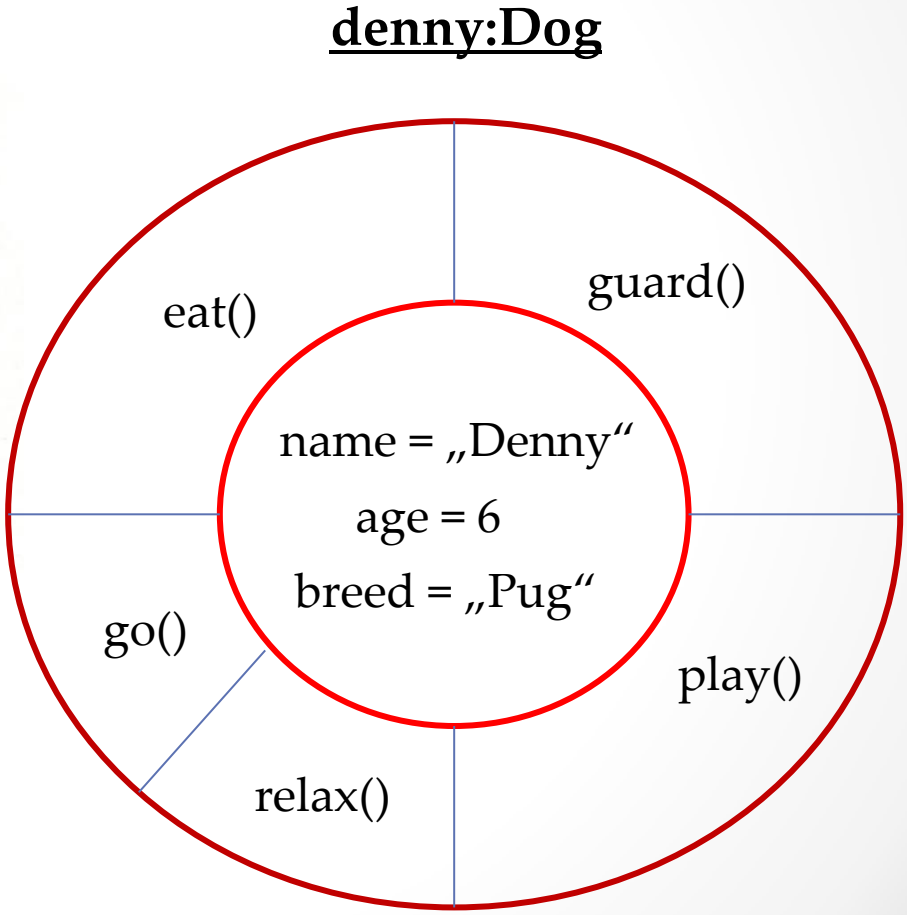
- Inheritance principle offers possibility to group objects according to their attributes/methods into higher wholes (parents) – generalisation
- Child class inherits all attributes/operations from the parent class and add some others attributes/operations that are not a part of the parent class
- Inheritance enables reusing in modelling and programming
- Clarity: it is not necessary to repeat a programming code
- Easy changes: changes in parent influences the child

Principle 3: Encapsulation … as the armour of the knight



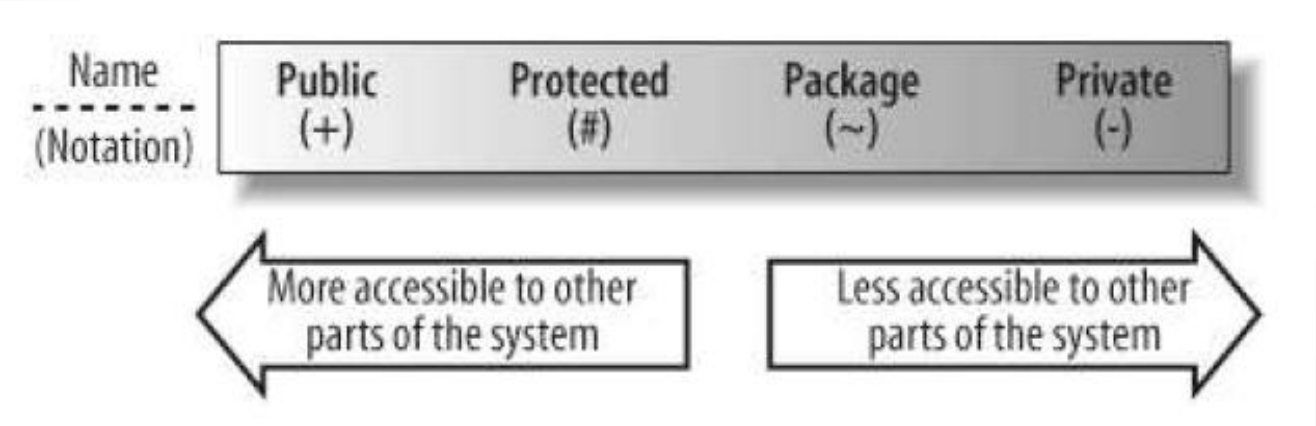
- Hidding internal details of the object against the outer environment
- It is the kind of protection against unauthorised interventions comming from the other objects or from the user
- Only the object itself determines what will be given from the internal state to the enviroment. This is done with the aid of methods
- Thanks to this the objects are not in states that do not correspond to the reality

Analogy



Parameters of visibility (1)

- **Once visibility characteristics are applied, you can control access to attributes, operations, and even entire classes to effectively enforce encapsulation.**



https://www.safaribooksonline.com/library/view/learning-uml-20/0596009828/ch04s03.html

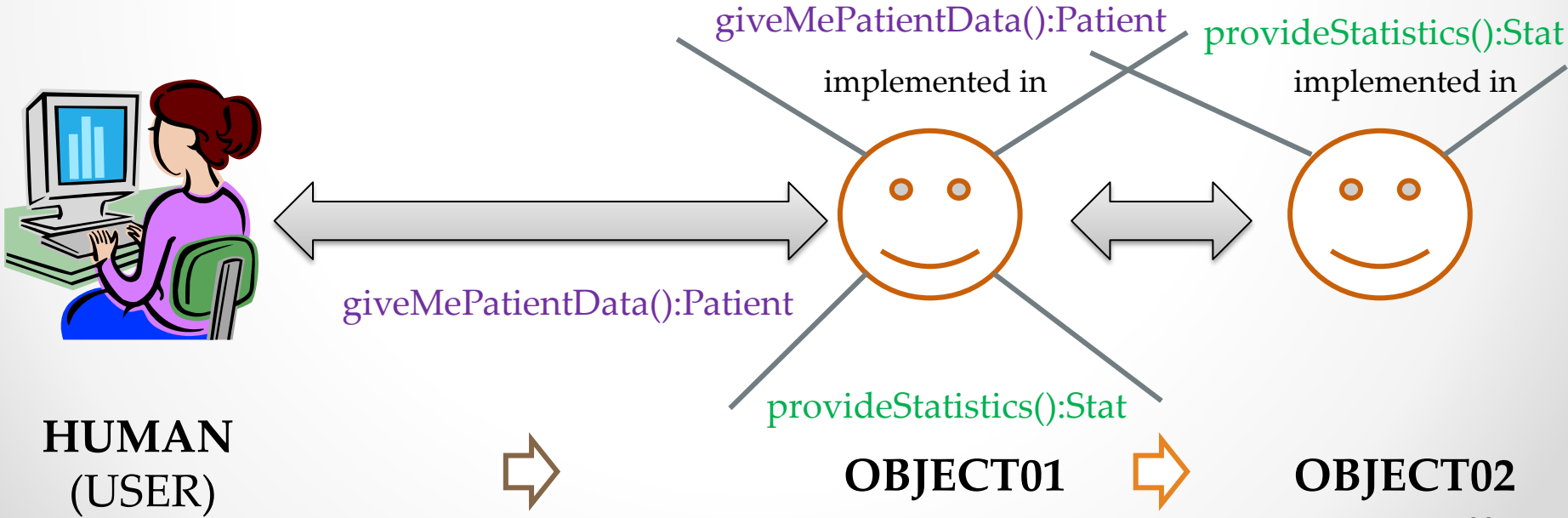
Parameters of visibility (2)

- **Public: Public the weakest visibility characteristic. Its notation is the plus symbol (+). Declaring a class attribute or method as public will make it accessible directly by any class.**
- **Private: Private is the strongest visibility characteristic. Its notation is the minus symbol (-). Declaring a class attribute or method as private will make it only accessible for the class itself.**
- **Protected: Protected is one the neutral visibility characteristic. Its notation is the hash symbol (#). Declaring a class attribute or method as protected will make it more visible than private attributes and methods, but less visible than public ones. In other words, protected elements can be accessed by a class that inherits from your class whether it is in the same package or not.**
- **Package: Package is the other neutral visibility characteristic. Its notation is the tilde symbol (~). Declaring a class attribute or method as package will make it visible only to any class in the same package. It doesn't matter if other class from other packages inherits from a class within the main package.**

https://weblogs.java.net/blog/felipeg/archive/2004/09/22/introduction-class-diagrams

Principle 4: Message sending

- Objects communicate together thanks to sending of messages
- Sending a message = request for execution of the particular operation



Relations between classes (object) Association

- Relations between classes have to exist for messages sending
- Association:
  - a linkage between two classes (they can send messages)
  - it is always assumed to be bi-directional (both classes are aware of each other)

Bi-directional association

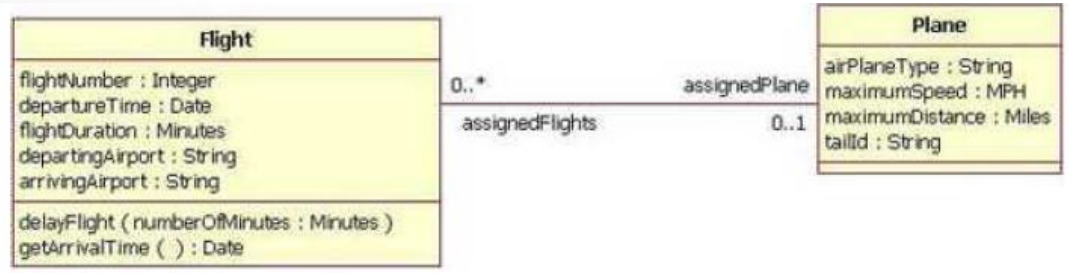
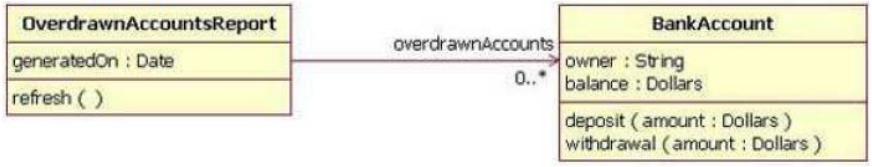


Figure 7: An example of a uni-directional association: The OverdrawnAccountsReport class knows about the BankAccount class, but the BankAccount class does not know about the association



Multiplicity

- **Multiplicity: how many objects can participate in the association**

Table 3: Multiplicity values and their indicators

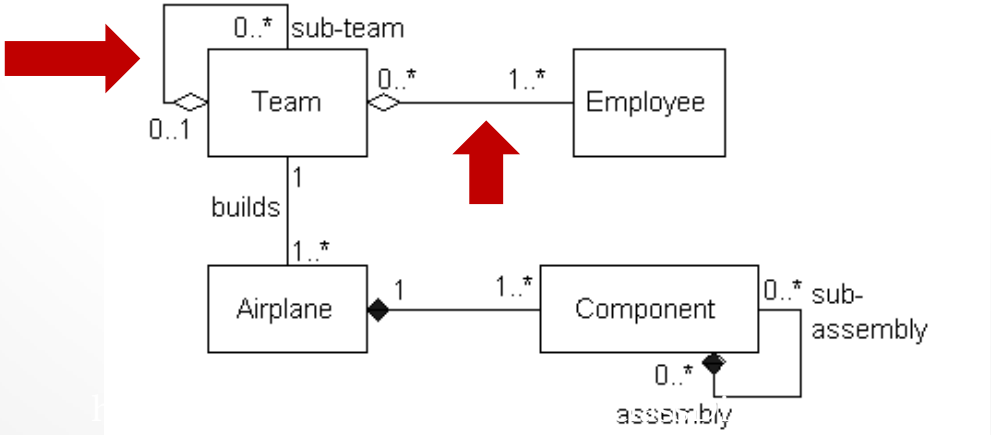
Potential Multiplicity Values

Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
*	Zero or more
1..*	One or more
3	Three only
0..5	Zero to Five
5..15	Five to Fifteen

* = infiniteness

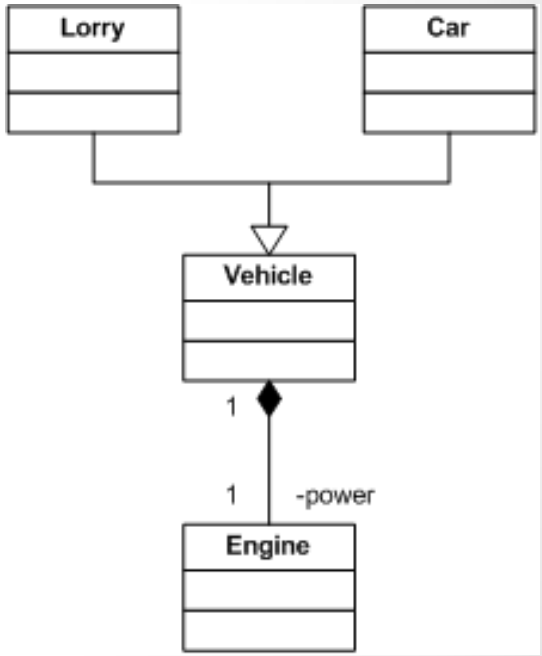
Relations between classes (object) Agregation

- **Agregation: An association with an aggregation relationship indicates that one class is a part of another class. In an aggregation relationship, the child class instance can outlive its parent class. Part of the whole can exist without the whole.**

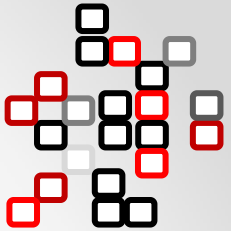


Relations between classes (object) Composition

- The composition aggregation relationship is just another form of the aggregation relationship, but the child class's instance lifecycle is dependent on the parent class's instance lifecycle.
- Figure below shows a composition relationship between a Company class and a Department class, notice that the composition relationship is drawn like the aggregation relationship, but a diamond shape is filled.



Polymorphism



- Polymorphism (from Greek πολύς, polys, "many, much" and μορφή, morphē, "form, shape")
- Polymorphism refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied in object-oriented programming and languages like the Java language
- Polymorphism ensures that we will be able to work with objects in the same way also in case if their behaviour is different

Polymorphism and analogy

- **Reaction on the same message is different for different objects (objects from various classes)**
- **Ability of methods to „work“ according to the type of an object onto which it influences**

Calling of one method = various reactions!

Let me your voice!
letMeVoice()

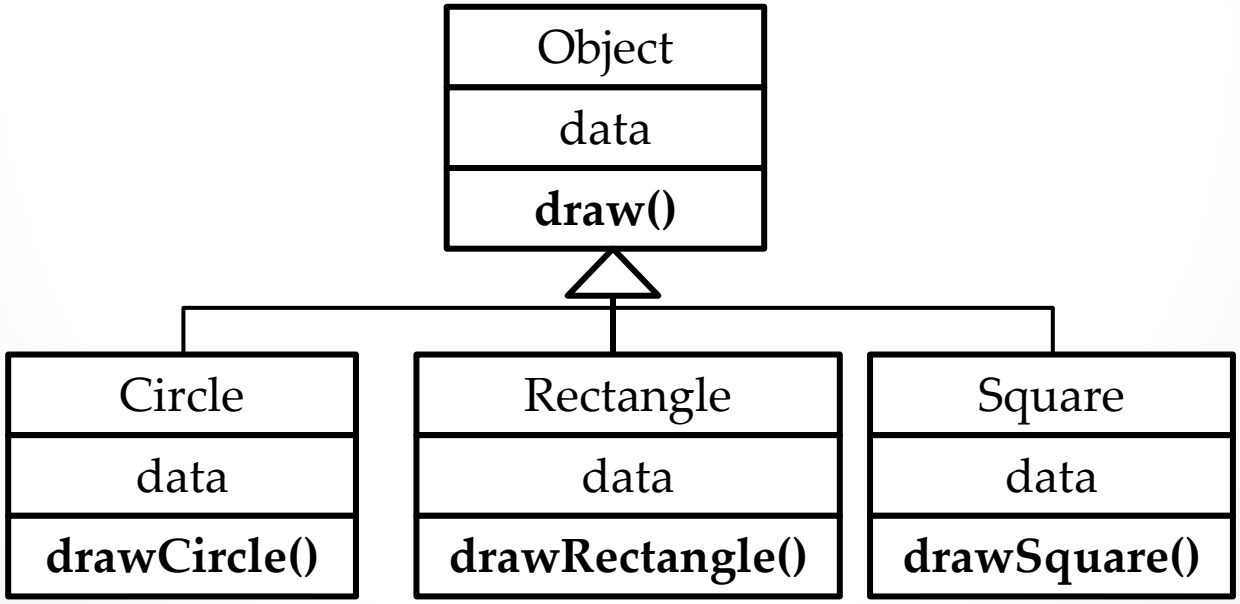
*letMeVoiceCat():**Mnau***

*letMeVoiceCat():**Haf***

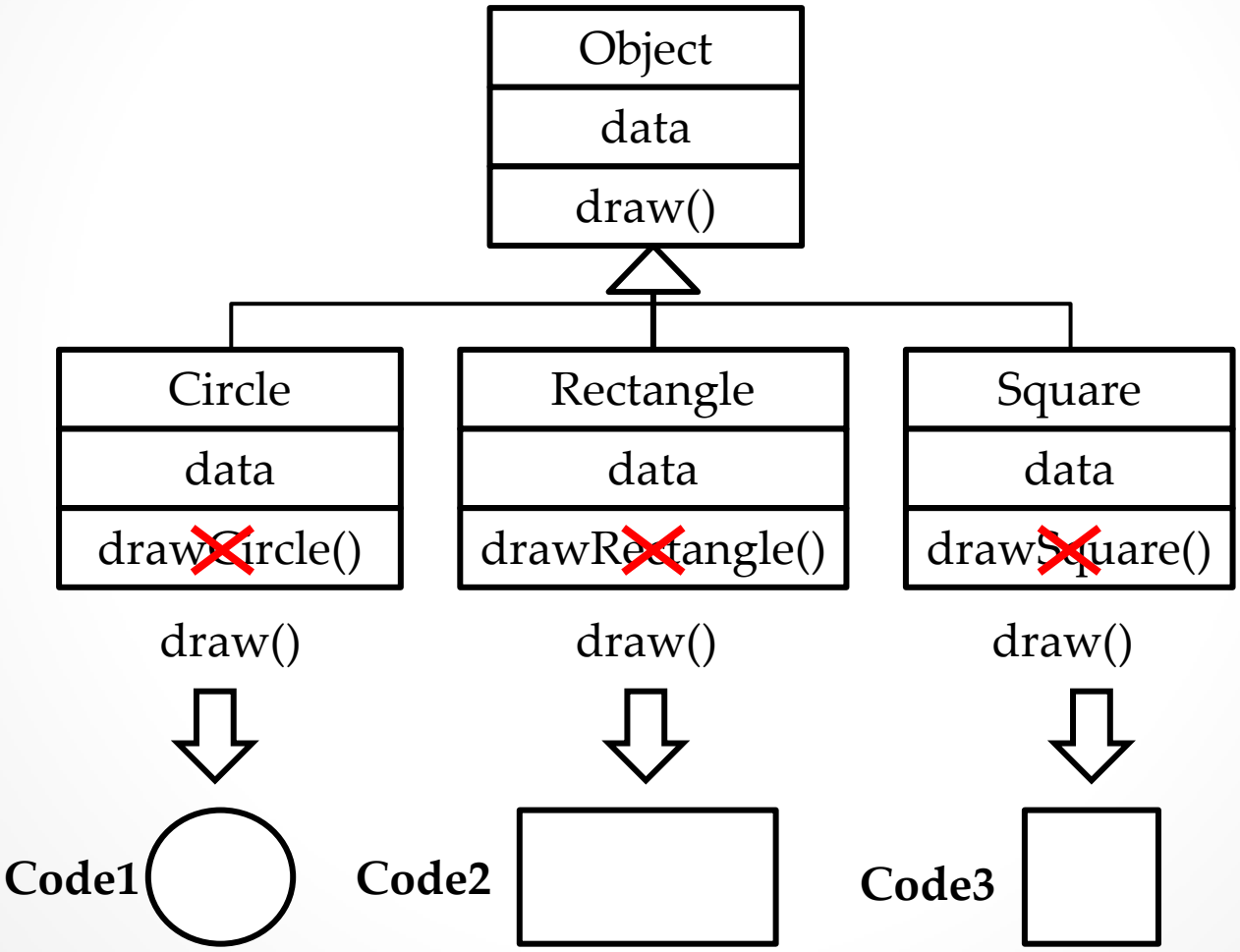
*letMeVoiceCat():**Íiíiááá***

Polymorphism

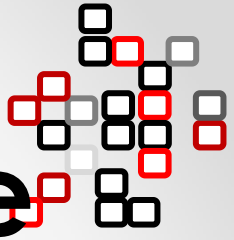
- **We will work with objects in the same way in cases if they have different behaviour**
- **Reducing complexity of the system**



Polymorphism



UML
Unified Modelling Language

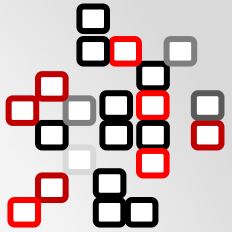


- **Standard for building object-oriented software applications**

The OMG specification states:

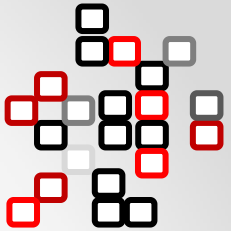
"The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components."

UML as conceptual language

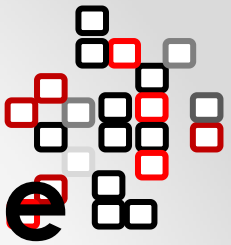


- It is not a methodology, it is the conceptual and visual modelling language providing notation for modelling
- It is graphically oriented rather than text oriented
- It focuses on conceptualisation and abstraction
- The UML evolved from the earlier works (OMT, OOSE, Booch) of „three amigos“ Booch – Jacobson - Rumbaugh

UML – history and present



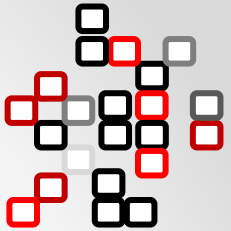
- Developed in the mid-1990's and standardised in 1997 (UML 1.1)
- UML 2.0 approved by OMG in 2003, released in May 2004
- Actual version = 2.5
- UML is now controlled by the Object Management Group (OMG)
- Variations and extensions exist
  - AML (The Agent Modelling Language): analysis and design of the multi-agent based systems



UML - possibilities for the usage

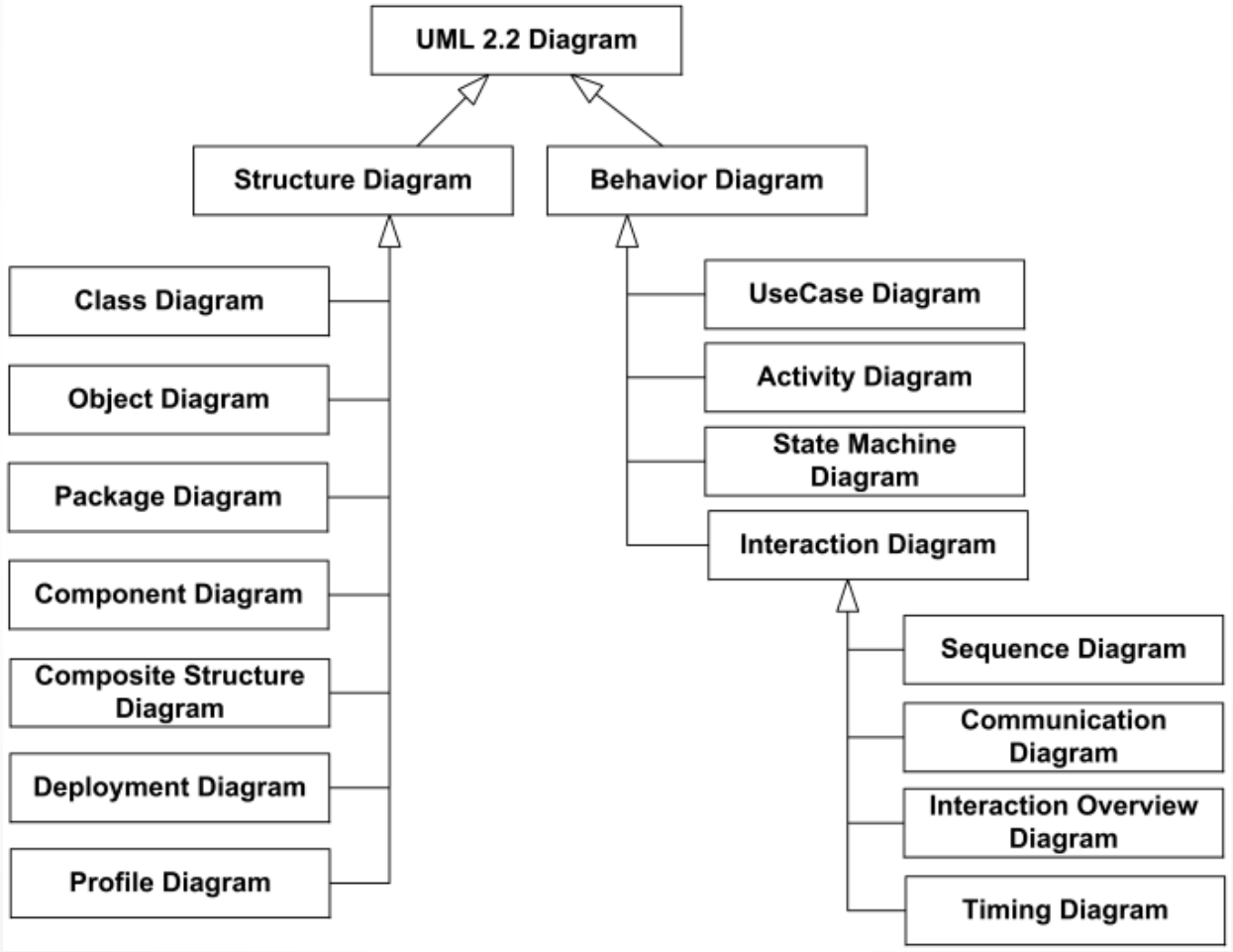
- UML as a sketch and whiteboard artifact
- UML as a blueprint for sotware analysts and architects
- UML as a sophisticated approach for code generation

Object-oriented programming languages



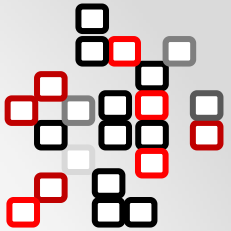
- Languages designed mainly for object-oriented programming:
  - C++
  - Java
  - C#
  - Python
- Languages with some object-oriented features:
  - Visual Basic
  - Fortran
  - Perl
  - PHP

Views on the real system
UML 2 diagram types



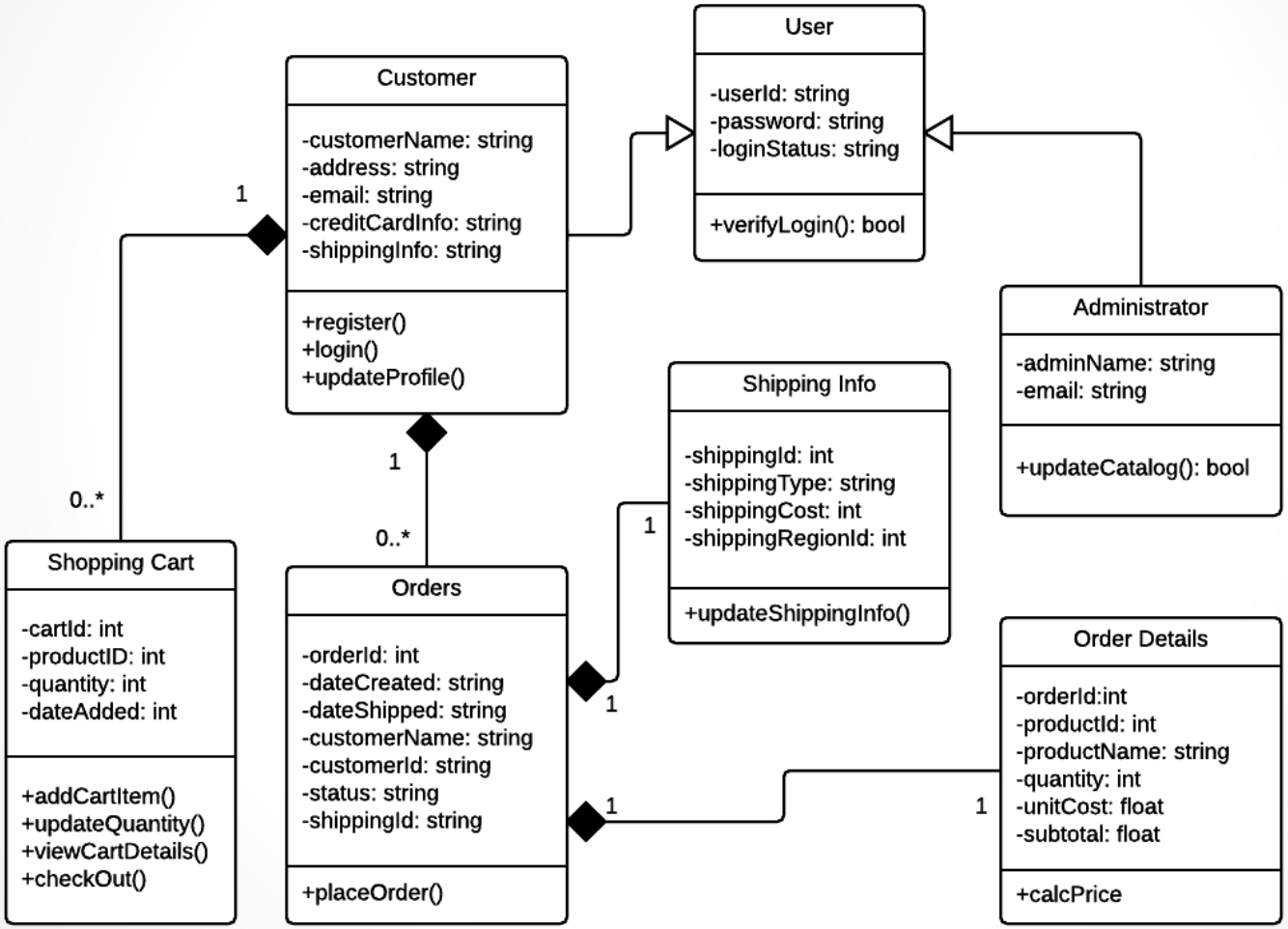
http://www.uml-diagrams.org/uml-22-diagrams.html

UML Class Diagram



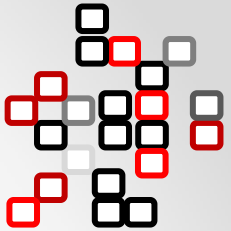
- The class diagram shows how the different entities (people, things, and data) relate to each other
- It shows a static structure of a system
- A class diagram can be used to display logical classes, which are typically the kinds of things the business people in an organization talk about — rock bands, CDs, radio play; or loans, home mortgages, car loans, and interest rates.
- It is used after use cases modelling

UML Class Diagram
Example: Online Shopping System

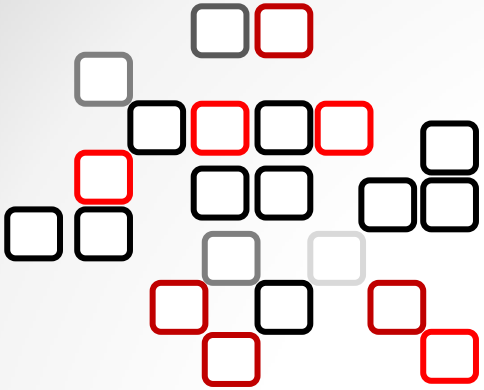


https://www.lucidchart.com/documents/view/[illegible]

Additional sources for your study



- **BOOKS**
  - **UML 2 and the unified process : practical object-oriented analysis and design / Jim Arlow and Ila Neustadt. ISBN 0-321-32127-8**
  - **System analysis design UML version 2.0 : an object-oriented approach / Alan Dennis, Barbara Haley Wixom, David Tegarden. ISBN 978-0-470-07478-7**
- **WEB sources**
  - **http://www.uml-diagrams.org/**
  - **http://www.tutorialspoint.com/uml/uml_interaction_diagram.htm**
  - **http://creately.com/diagram-type/use-case**
  - **http://www.cs.toronto.edu/~sme/CSC340F/slides/11-objects.pdf**
  - **http://wps.prenhall.com/wps/media/objects/14735/15089538/M13_HOFF2253_11_SE_C13WEB.pdf**



THANK YOU FOR YOUR ATTENTION!