CSIT121
Object-Oriented Design and Programming

Dr. Fenghui Ren

School of Computing and Information Technology
University of Wollongong

Lecture 2 outline

- Object-oriented concept
- Class design
- Encapsulation
- Basic Unified Modelling Language (UML)
- Class design with the UML class diagram

Focus of this subject

This subject is not only about Python

• If know a programming language and you know how to use computers, it does not mean that you can develop software systems

If you know alphabet and you can type quickly, this may not be enough to write a novel

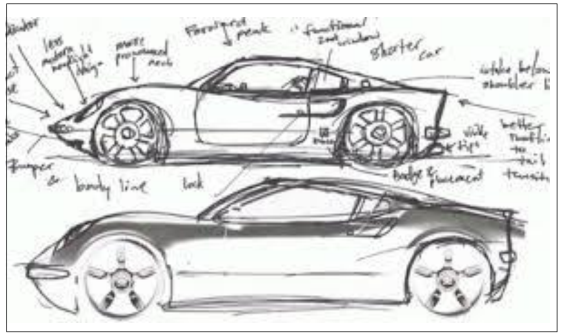
To become a programmer

- you need to be familiar with advanced software development concepts
- you need to know how to efficiently utilize these concepts using appropriate programming languages

Software development challenge

Software development has unique challenges which are not common for other industries

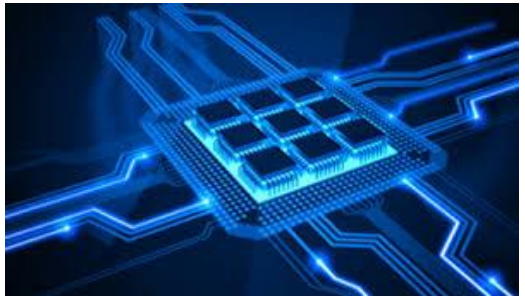
- When you design a car you can foresee how it will look like when it is made
- Reverse engineering can reproduce car design documents (identical to original ones)



However…

- When a program is executed, it is a sequence of invisible electrical signals which represent binary microprocessor instructions
- It is practically impossible to reproduce the original program code from binary instructions

How to design programs when you can't see the final product?



Imperative Programming Concept

The oldest concept where a program consists of a sequence of commands for the microprocessor to execute

Example: Assembly language is a symbolic representation of microprocessor instructions with one-to-one mapping

```
pushl %ebp
movl %esp, %ebp
addl 8(%ebp), %eax
```

$\longleftrightarrow$

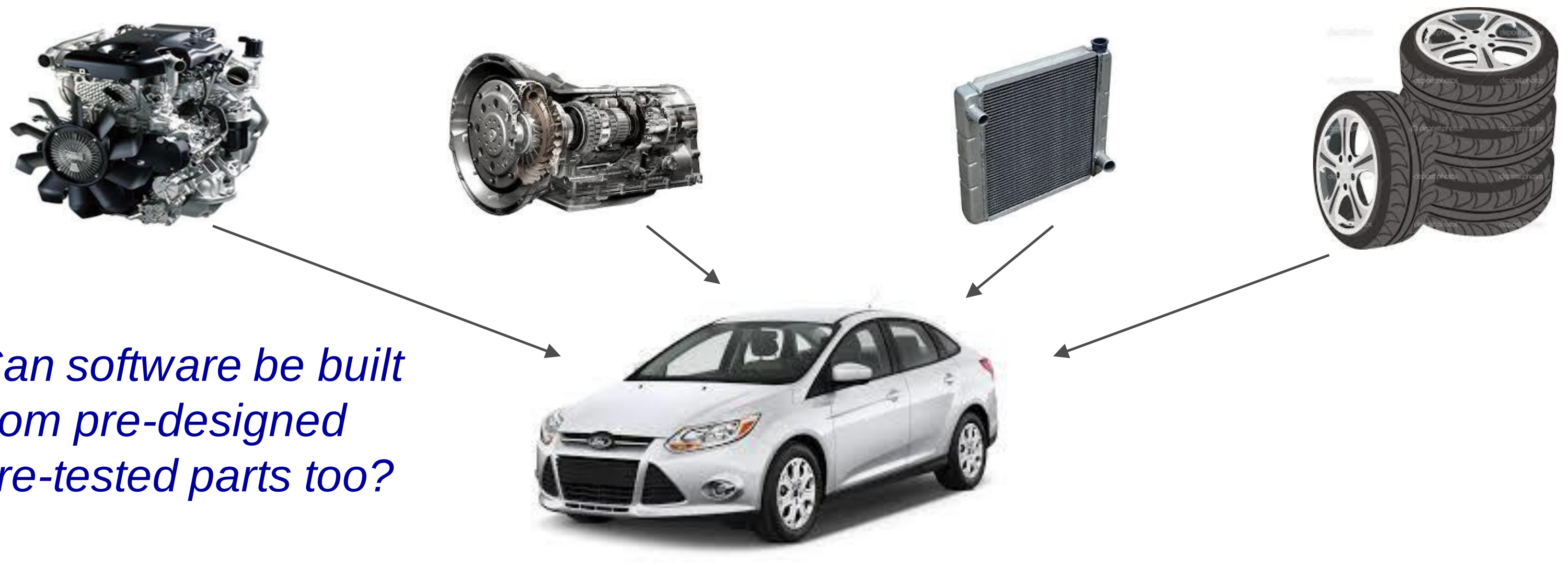
```
11010011
01010010
10000011
```

How to increase productivity of software development?

• When you virtually have to 'live in the microprocessor' and think like a machine to write even a simple program, you cannot be productive
• It is close to impossible to reuse assembly code. You need to start every new project practically from scratch

How to find an efficient design methodology ?

We could try to borrow ideas from other industries

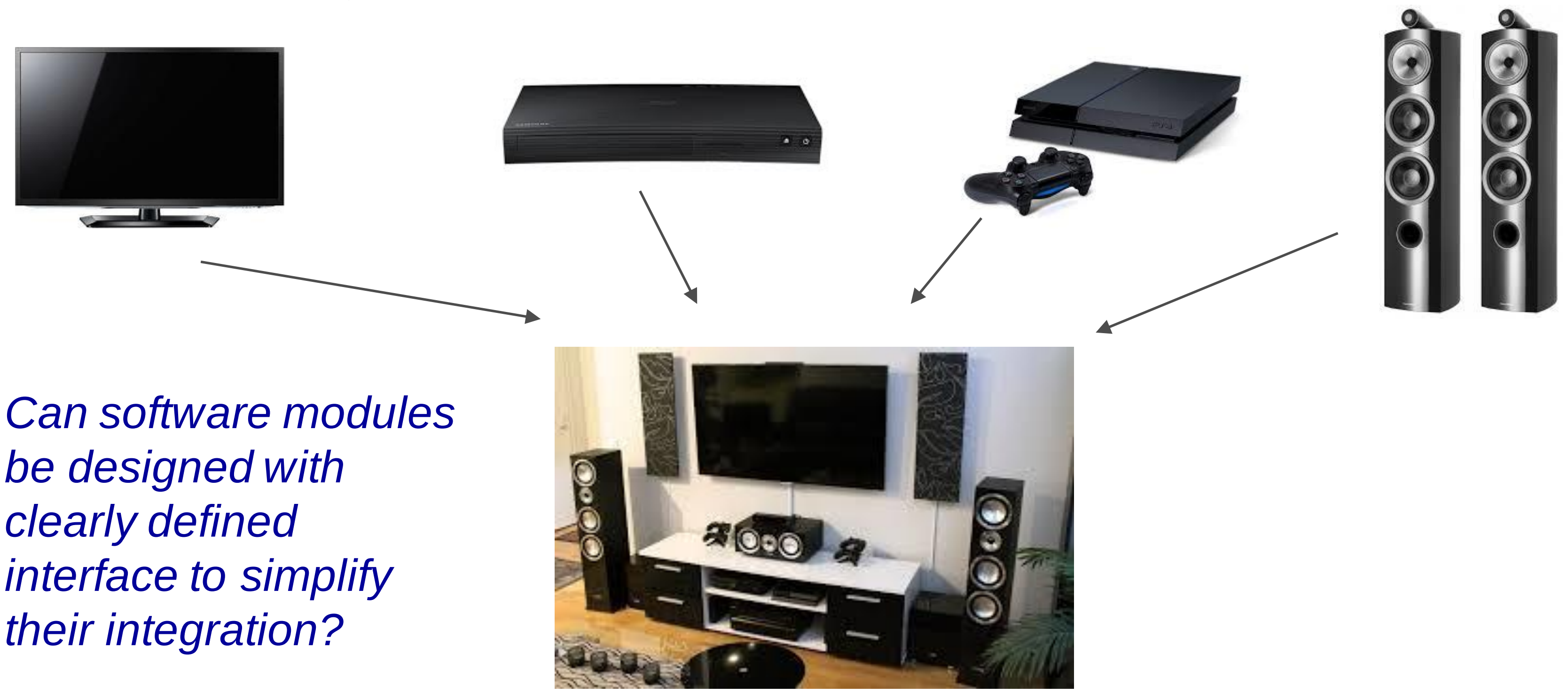


Can software be built from pre-designed pre-tested parts too?

- Cars are made of preassembled parts
- To build a new car you don't need to design every part from scratch – some parts can be reused from previous models, or be purchased from new suppliers without making any changes in their internal structure

How to find an efficient design methodology ?

We could try to borrow ideas from other industries

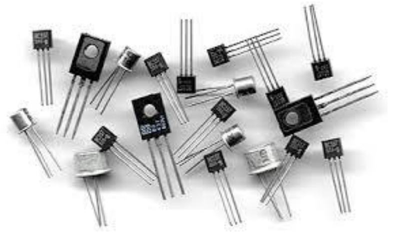


Can software modules be designed with clearly defined interface to simplify their integration?

- To assemble a home theatre you can buy modules from different manufacturers according to your needs
- They can be easily plugged together as their interfaces are standardized

How to find an efficient design methodology ?

Other examples:



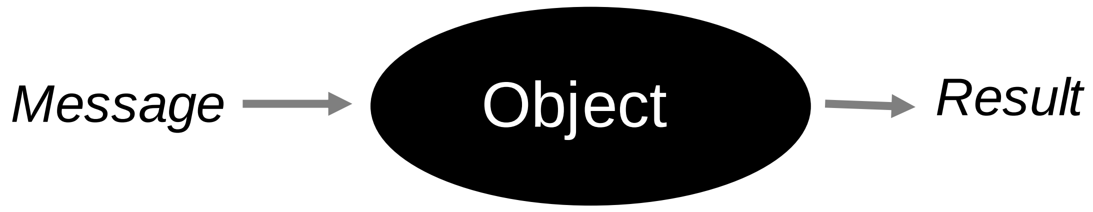
From transistors to integrated circuits

From bricks to pre-fabricated panels

- Transition to pre-fabricated generic reusable modules with well defined interface components has been a common trend for all industries
- To increase productivity, computer programming had to follow the same trend
- Research on efficient programming concepts led to development of the **Object Oriented Design** methodology in the late 60th

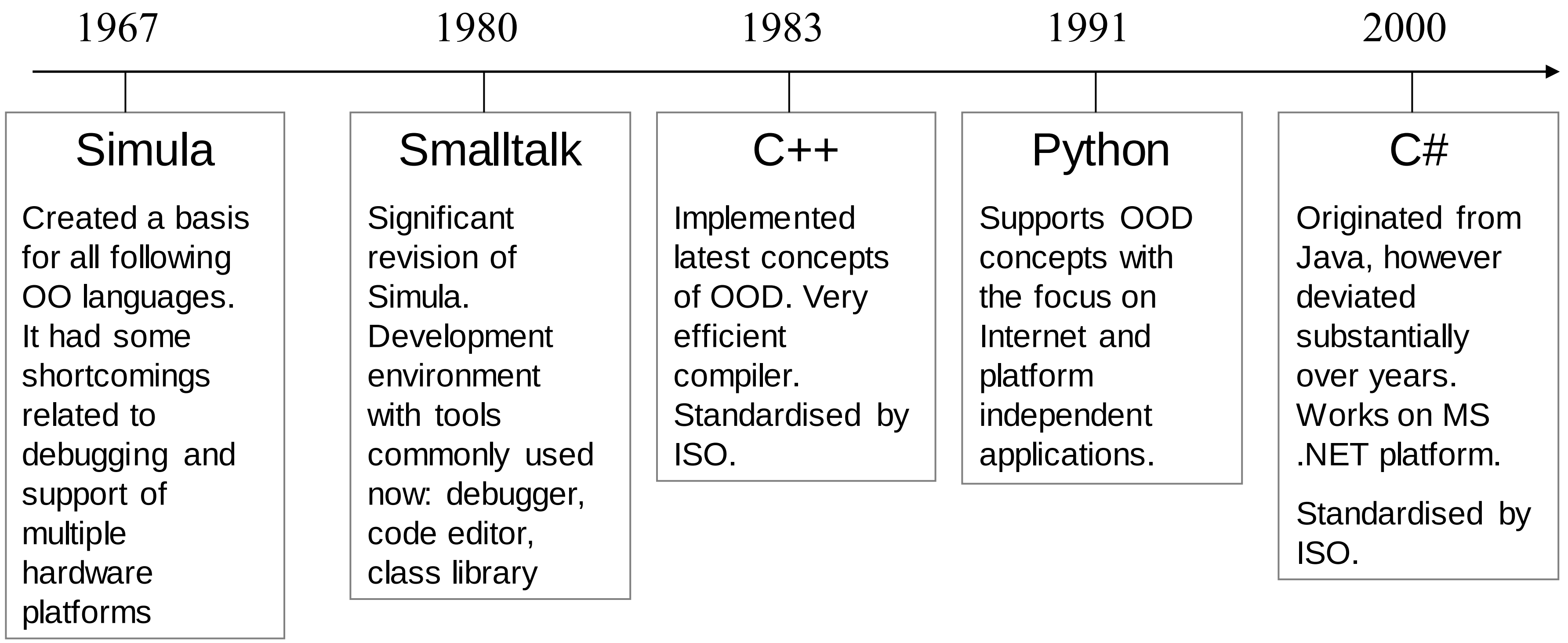
Objects

- Object Oriented Design methodology introduces the concept of **objects**
- Like mechanical and electrical parts in other industries, **objects become major software building blocks**
- Rather than think about commands and microprocessor instructions, you need to think about objects, their **properties**, their **behaviors** and how their **interactions** can implement your program functionality
- In general, objects can receive messages from other objects and provide responses. To use them, you don't need to worry about their internal structure. It is hidden.



Evolution of OOD languages

- The OOD methodology has substantially evolved since it was introduced in the 60th
- Evolution of OOD programming languages reflects the evolution of OOD programming concepts



1967 — **Simula**

Created a basis for all following OO languages. It had some shortcomings related to debugging and support of multiple hardware platforms

1980 — **Smalltalk**

Significant revision of Simula. Development environment with tools commonly used now: debugger, code editor, class library

1983 — **C++**

Implemented latest concepts of OOD. Very efficient compiler. Standardised by ISO.

1991 — **Python**

Supports OOD concepts with the focus on Internet and platform independent applications.

2000 — **C#**

Originated from Java, however deviated substantially over years. Works on MS .NET platform.

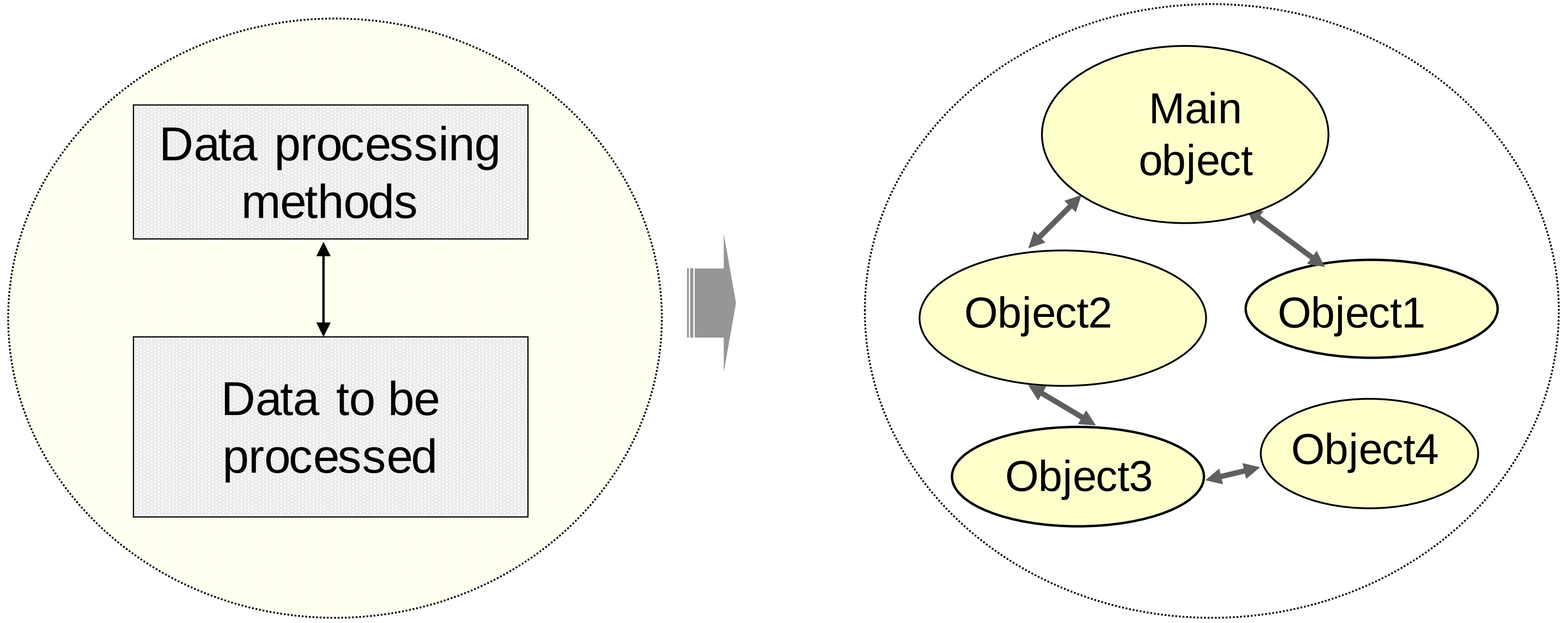
Standardised by ISO.

Object Based Design

- The main purpose of any program is to process data
- The major task of OOD is how to split system's **data** and **behaviour** into a set of interacting objects

A functional view of the program

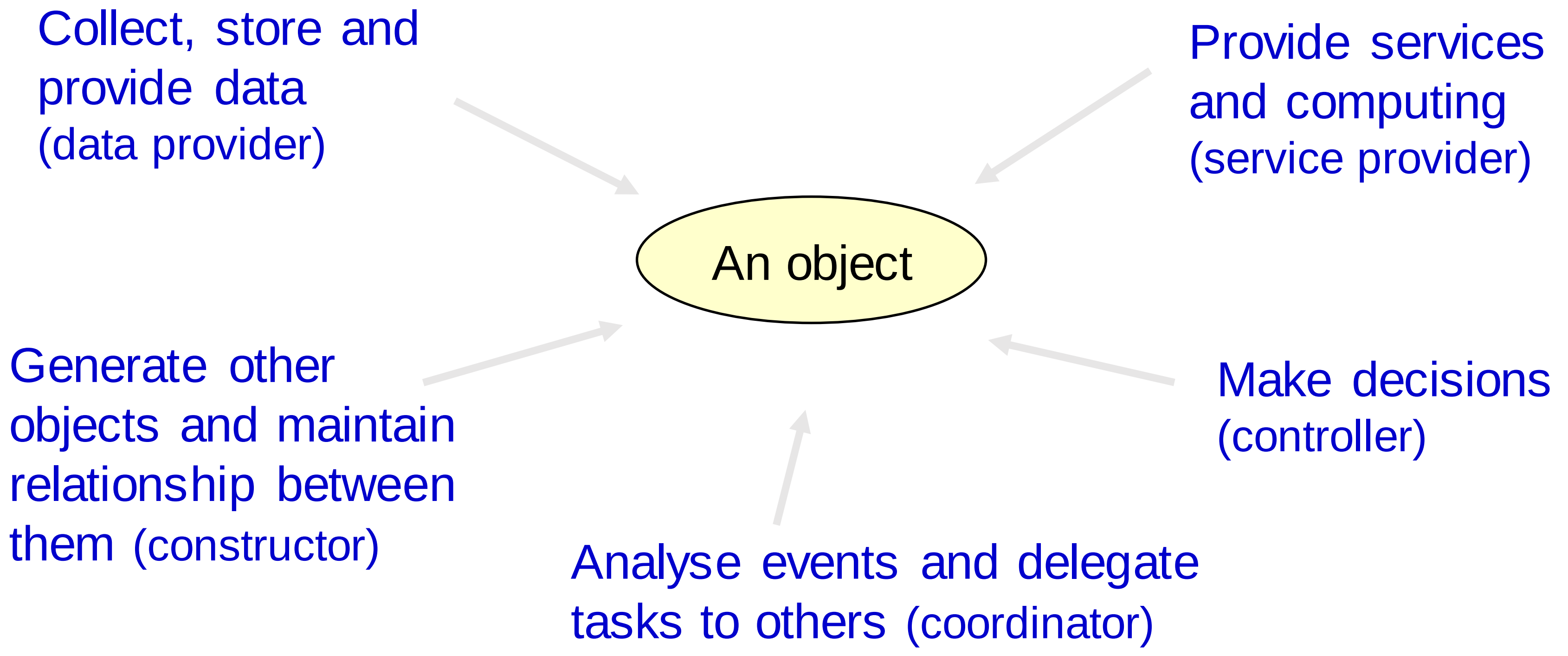
An OOD view of the program



What can an object do ?

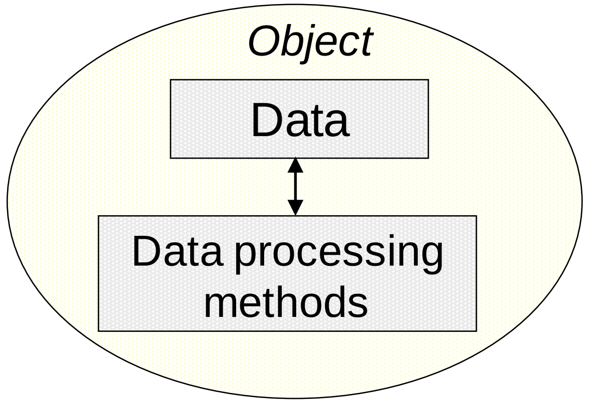
- When you describe your program as a collection of interacting objects, you need to have a clear idea what role the objects will play in your application
- An object may play just one role, or several typical roles

Role Stereotypes


Collect, store and provide data (data provider)
Provide services and computing (service provider)
An object
Generate other objects and maintain relationship between them (constructor)
Make decisions (controller)
Analyse events and delegate tasks to others (coordinator)

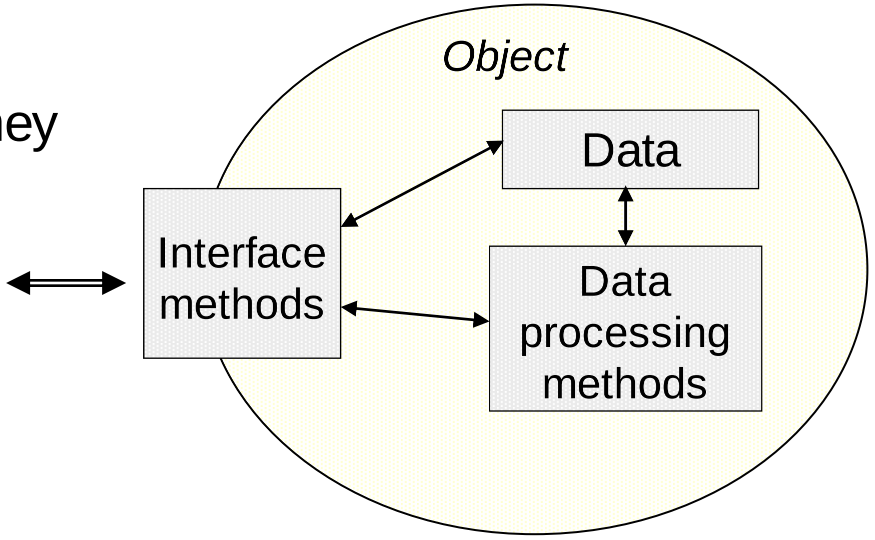
Internal structure of objects

- Each object implements a certain part of the system functionality. Thus, it should contain **data** and **methods** to process this set of data



How can objects interact?

- To interact with other objects, they also include interface methods
- Interaction between objects is possible **only through interface methods**



What does an object look like?

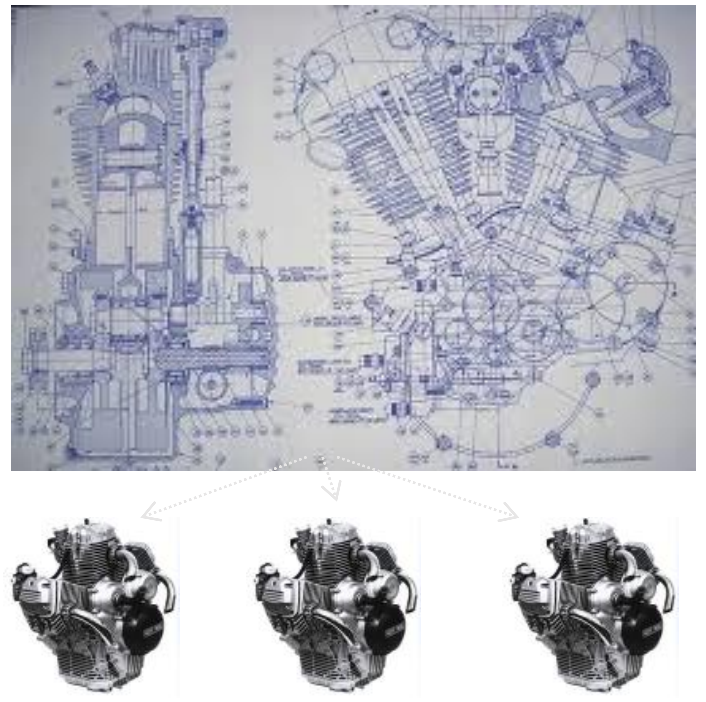
- **Physically**, it consists of binary data and microprocessor instructions, which occupy a reserved block of memory
  (its actual layout is system dependent)
- Links between objects can be established at the compilation time or dynamically at run time

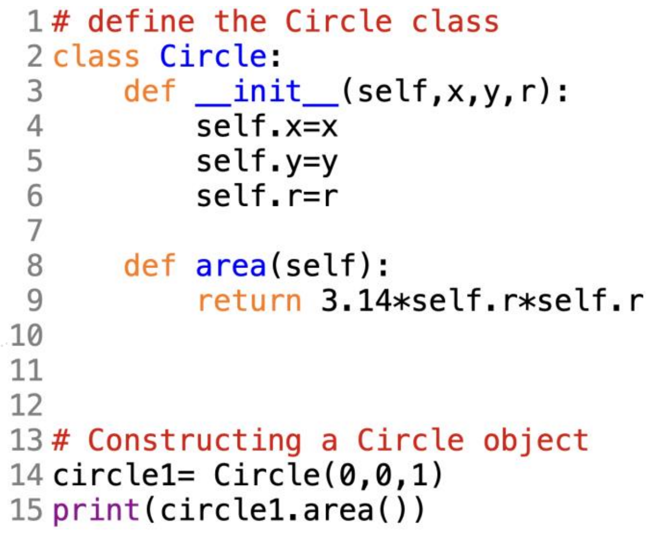
- **Logically**, it must be declared in your program
- When an object is declared, it is given a unique name
- Looking through a program code, you can see all objects declared there and follow the logical relationship between them

How can objects be created ?

Parts of mechanical systems are made based upon their description – blueprints

Objects of a software system are generated based upon their description – classes





```python
# define the Circle class
class Circle:
    def __init__(self,x,y,r):
        self.x=x
        self.y=y
        self.r=r

    def area(self):
        return 3.14*self.r*self.r



# Constructing a Circle object
circle1= Circle(0,0,1)
print(circle1.area())
```

Instantiation

circle1 circle2 circle3

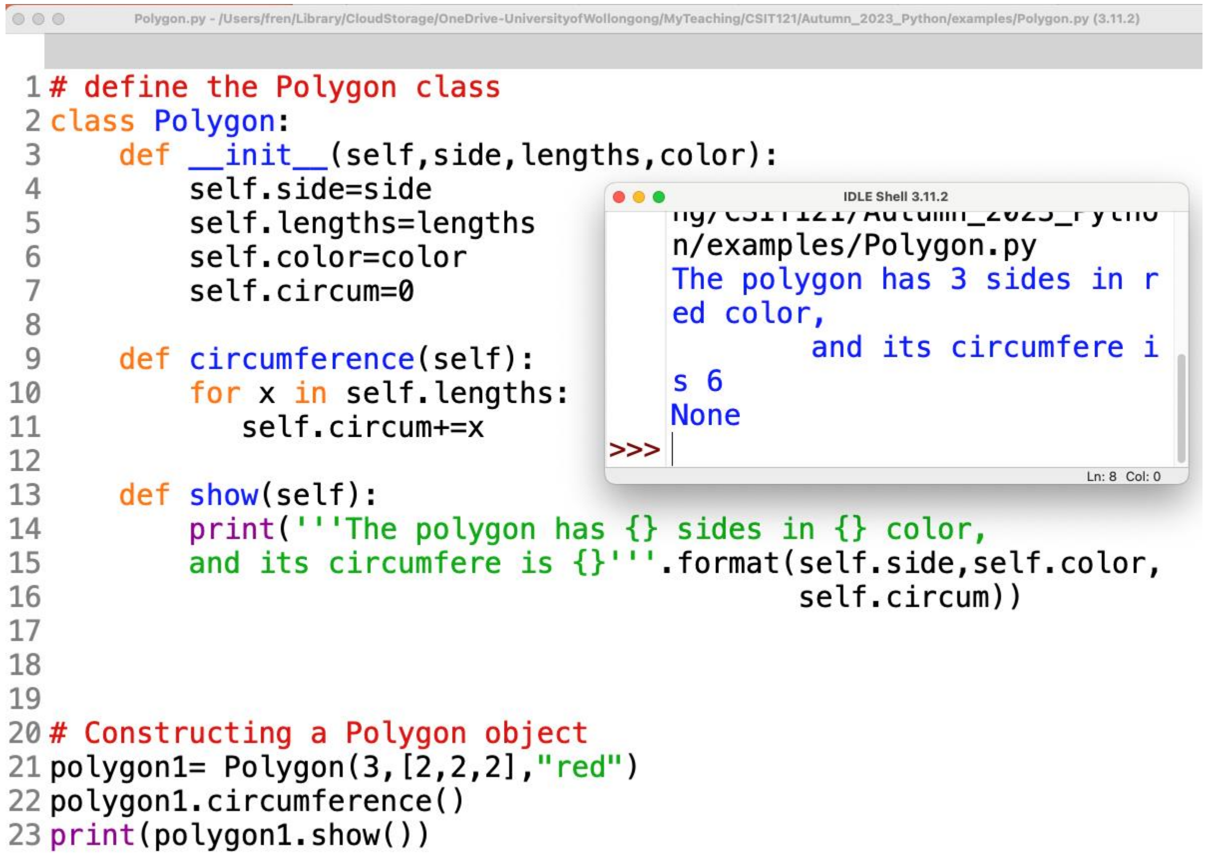
- A class is an abstraction defined by a programmer but NOT placed in the computer memory
- An object is an instance generated and placed in computer memory (many similar objects can be generated from one class)

Classes

- Classes can be interpreted as blueprints, or prototypes for objects
- You can't use classes directly, you only can use objects created based on their classes.
- Objects are not necessarily clones even when they are created from the same class

For example: a class `Polygon` can be used to create triangles and rectangles as instances (objects)

- To generate different objects, we may specify different properties such as **number of sides**, the **lengths of those sides**, their **colors**
- The object behaviour are things which require actions - how the **area**, or the **circumference** are calculated, how the object is displayed, etc



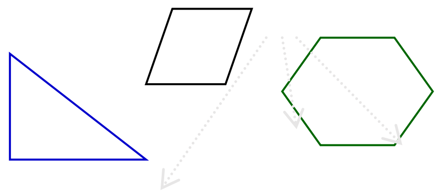
Polygon.py - /Users/fren/Library/CloudStorage/OneDrive-UniversityofWollongong/MyTeaching/CSIT121/Autumn_2023_Python/examples/Polygon.py (3.11.2)

```python
# define the Polygon class
class Polygon:
    def __init__(self,side,lengths,color):
        self.side=side
        self.lengths=lengths
        self.color=color
        self.circum=0

    def circumference(self):
        for x in self.lengths:
            self.circum+=x

    def show(self):
        print('''The polygon has {} sides in {} color,
        and its circumfere is {}'''.format(self.side,self.color,
                                          self.circum))



# Constructing a Polygon object
polygon1= Polygon(3,[2,2,2],"red")
polygon1.circumference()
print(polygon1.show())
```

IDLE Shell 3.11.2

```
n/examples/Polygon.py
The polygon has 3 sides in r
ed color,
        and its circumfere i
s 6
None
>>>
```

Ln: 8 Col: 0



Classes

- Conceptual class (start):
  - OOA
  - From your mind
  - A rough description of items and concepts
  - Can only understood by yourself
- Readable class (through):
  - OOD
  - A formal description of classes
  - Data structure, functions, relationships
  - Can be understood by anyone with the background knowledge
  - UML Class Diagram
- Executable class (target):
  - OOP
  - Programs
  - Can be used by everyone
  - Create objects

Summary: Class vs. Object

- A class is a template for objects, and it is a logical entity.
- A class defines properties and functions, such as data type, structure, range, and requirements and behaviors of methods.
- A class does not contain the actual values, and just act as a container.
- By considering the difference between actual values of a class, one class can have multiple different objects.
- A class is declared only once, and no memory is allocated for a class. Therefore, a class can not be executed.

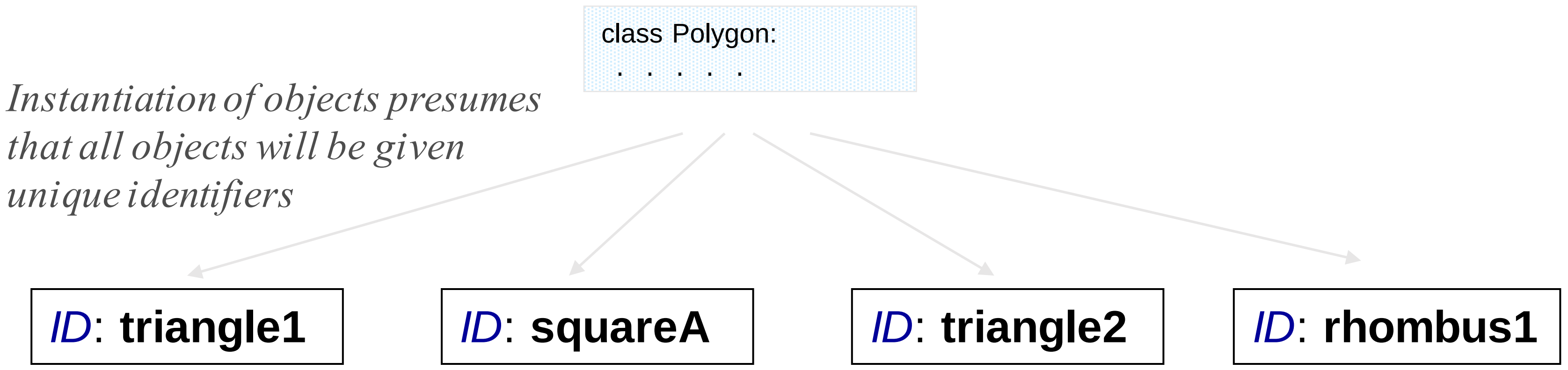
Summary: Class vs. Object

- An object is an instance of a class, and it is a physical entity.
- An object has a state in which all of its properties MUST have values.
- Objects can be created many times, and each object will be allocated memory once created. Objects created from the same class will have different locations in the computer memory.
- Objects are executable. However, the objects of the same class may act very differently.

Object features

1. **Identity**

- Correct interaction between objects may not possible without reliable identification of objects
- Each object has a unique identifier that is used to recognize it



class Polygon:
. . . . .
Instantiation of objects presumes that all objects will be given unique identifiers
ID: triangle1
ID: squareA
ID: triangle2
ID: rhombus1

Identity (a unique name) of an object can be:

- specified by a programmer
- assigned by another object that takes a role of constructer

Object intrinsic features

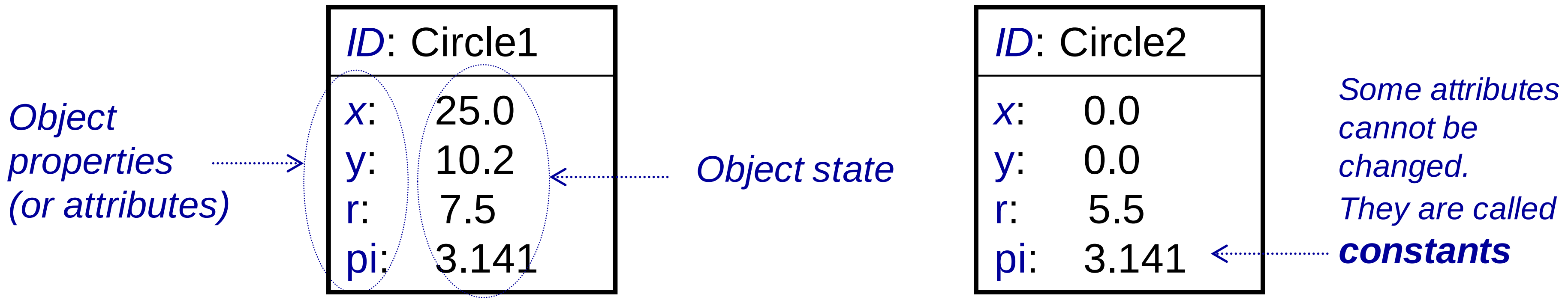
2. **State**

- Object-Oriented concept presumes that objects store data
- all data are stored with the purpose to reflect properties (or attributes) of objects

Example properties of a circle: (x,y) coordinates of the center, r radius

properties of a bill: due date, amount to pay

- A set of actual values stored in an object is referred as an object state

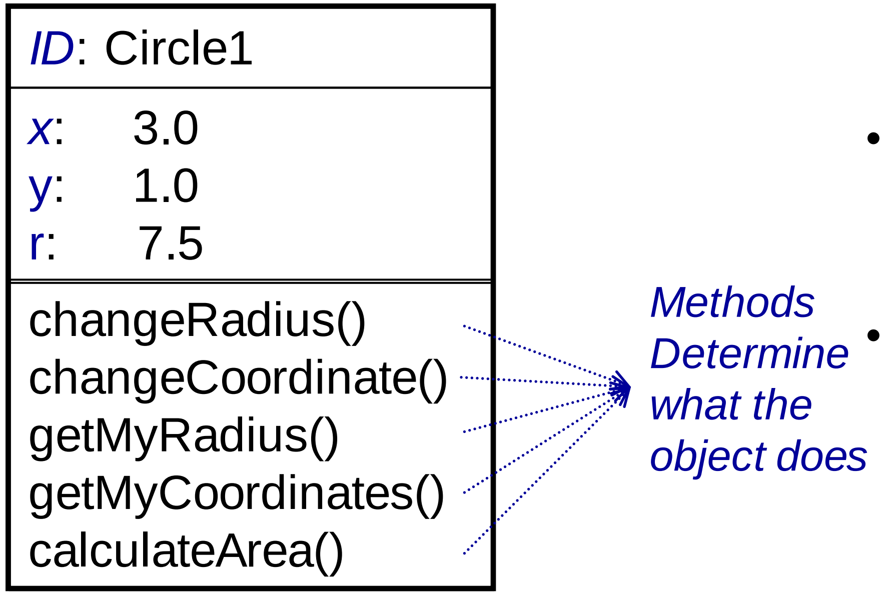

Object properties (or attributes)
ID: Circle1
x: 25.0
y: 10.2
r: 7.5
pi: 3.141
Object state
ID: Circle2
x: 0.0
y: 0.0
r: 5.5
pi: 3.141
Some attributes cannot be changed.
They are called **constants**

- Objects instantiated from the same class will have
  - different IDs,
  - the same attributes,
  - the same or different states

Object intrinsic features

3. **Behaviour**

- Instantiated objects are expected to act and react according to their roles (collect and store data, make decisions, provide services, etc)
- A set of supported actions defines object behaviour (what it does)
- Actions in OOD are called **methods**



- Some methods can change the object state (changeRadius, …).

  These methods are called **mutators**
- Some other methods can only access the value of an attribute without changing the object state (getMyRadius, …)

  These methods are called **accessors**.

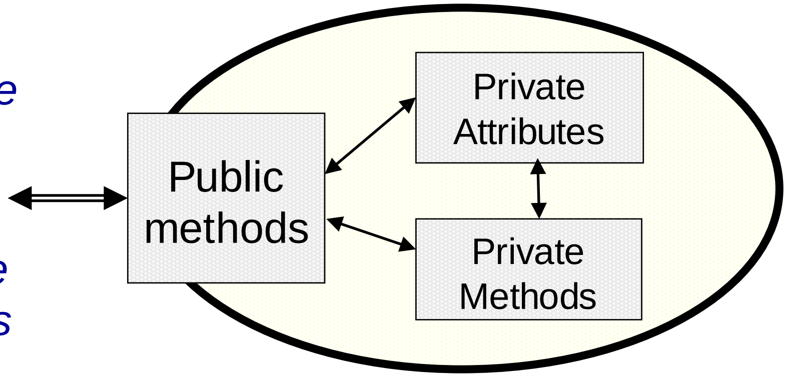
Fundamental OOD concepts

Encapsulation

- Some industrial products are sealed to prevent access to their internal components. They can be used only through external terminals
- OOD stipulates that to prevent unauthorized access to the most critical object attributes and methods, they need to be placed into a protective wrapper inside the object to make sure that:
  - well designed and tested objects cannot be alternated or corrupted by other poorly designed objects
  - hidden components can be modified without affecting object interaction (providing that interface methods are not affected by modifications)

*• Attributes and Methods which are specified as **Private** are hidden inside objects and cannot be accessed directly from outside*

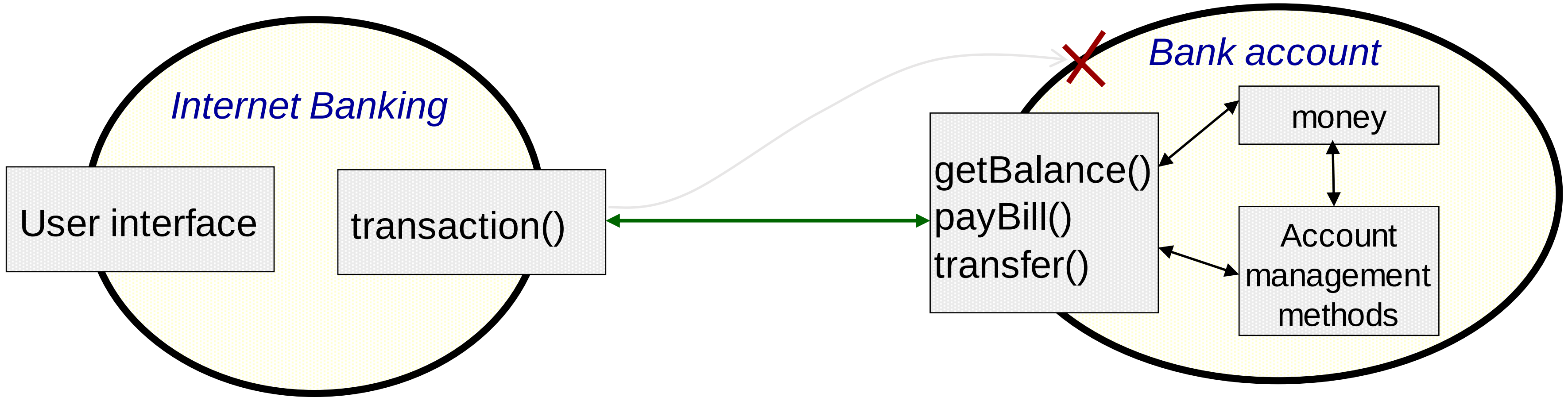
*• Methods specified as **Public** can be used for interaction with other objects*



Fundamental OOD concepts

Encapsulation

- According to OOD, interaction between objects can take place only through public methods
- In general, all attributes should be specified as private. If they need to be accessed from outside, this should be possible only through a limited set of public methods (Accessors or Mutators)
- Methods which are irrelevant to object interaction also should be private
- If a private attribute, or a private method can be accessed directly from outside, this indicates a serious design oversight (a safety bug)



Quiz

You need to implement an object Clock that

- counts time
- provides the current time on request
- can change on request the output format from 24hrs to AM/PM
- can set alarm
- can make beeping sound when alarm is on

	ID: Clock1
	hours: 12 minutes: 25 seconds: 34
+ acc	getTime()
+ mut	setOutputFormat()
+ mut	setAlarm()
- acc	countTime()
- acc	makeSound()

1. Which methods should be public and which ones should be private ?

2. Which methods are accessors and which ones are mutators?

Verbal description, ambiguity, confusion,…

- You may try to describe your program using a natural language
- Natural languages do not have exact rules for technical descriptions
- Even a simple natural language description can be interpreted differently by different listeners (the more details you provide, the higher chance for misinterpretation)
- A formal modelling language is needed to describe software designs unambiguously

UML is a formal tool for software systems modelling

Object Oriented Methodology

- OOD is not just a basis for OO programming languages, but a way of looking at the whole of the software development process – OO Methodology
- Making a program comprised of interacting objects is a big design challenge that cannot be done in a hurry
- OOM presumes that a lot of work needs to be done before you get down to typing your code. OOM prescribes a formal multistage process
- A program should be formally described using the UML (Unified Modeling Language) at the system design stage
- A UML description can be implemented using different OO programming languages

What is UML and how can it help?

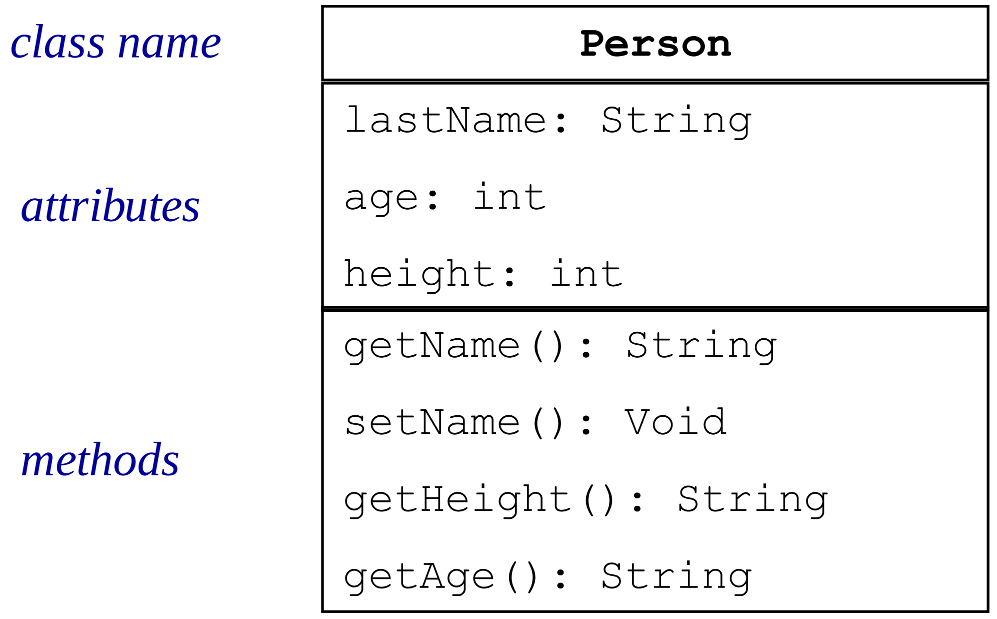
- UML (Unified Modelling Language) is a consolidation of research on OO modelling
- UML was accepted by the ISO as an international standard in 2000
- UML can be used
  1. To make sketches when you want to communicate key points of your program architecture
  2. To provide a formal and detailed specification of a software system
  3. To generate code directly from UML diagrams if a special tool is used
- UML models cannot be misinterpreted
- UML is scalable and can be used for small and big projects
- UML facilitates the software development process and simplifies debugging

UML Class Diagrams

- Class diagrams:
  - Contain classes and relations between them
  - Specify class **attributes** sufficient to describe *any* state of the object instantiated from a class
  - Specify **methods** sufficient to implement a required object behaviour
  - Specify **object state**

UML class symbol

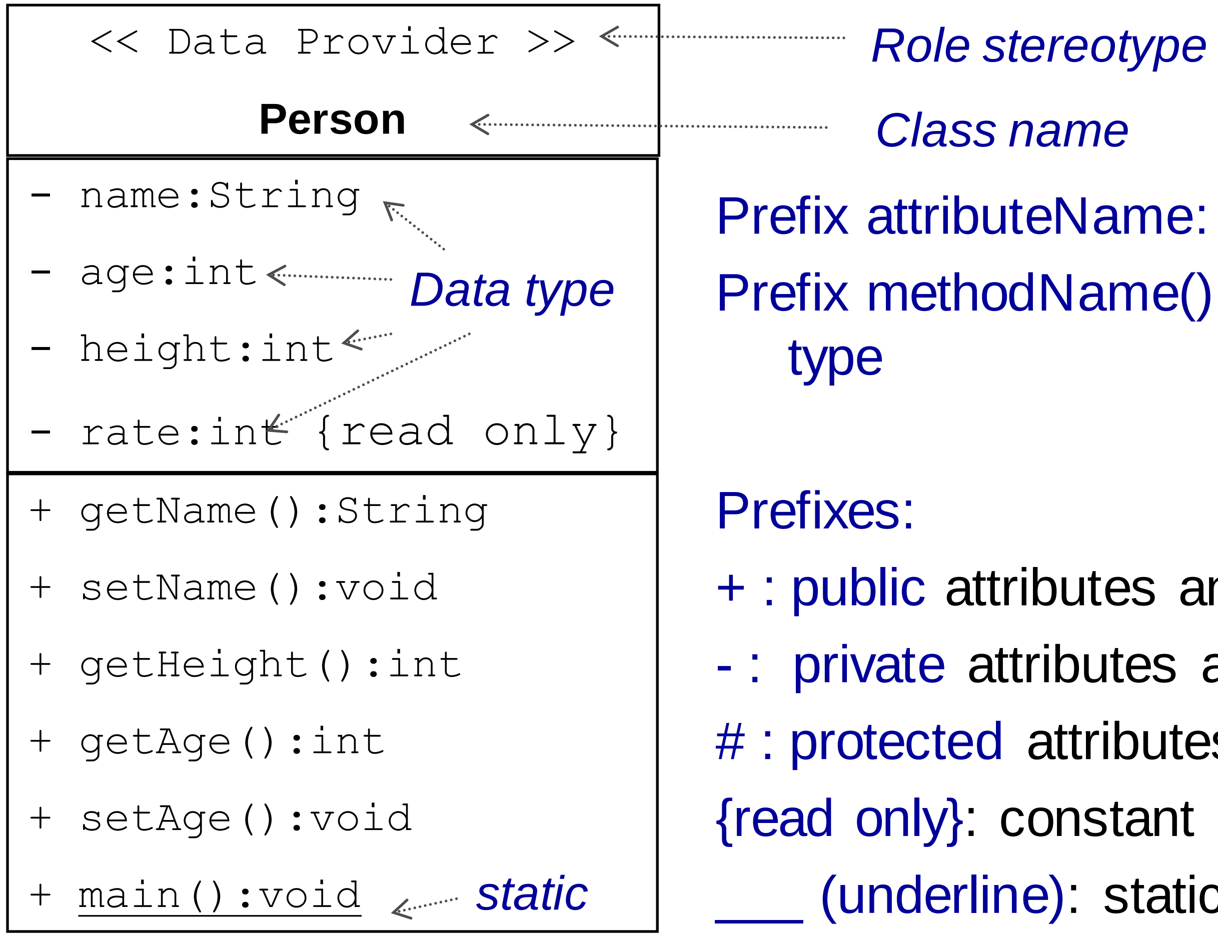
- The most basic class symbol is a box with the class name. It is used when class properties are not important (usually at initial stages of system design)
- Adding in more details may be needed at later stages
  - Attributes, or data types
  - Methods



- The level of details depends on who the diagram is intended for
- Class symbols are the building blocks of class diagrams

UML class symbol

- Role stereotypes
- Visibility prefixes



Prefix attributeName: Data type

Prefix methodName(): Return data type

Prefixes:

+ : public attributes and methods

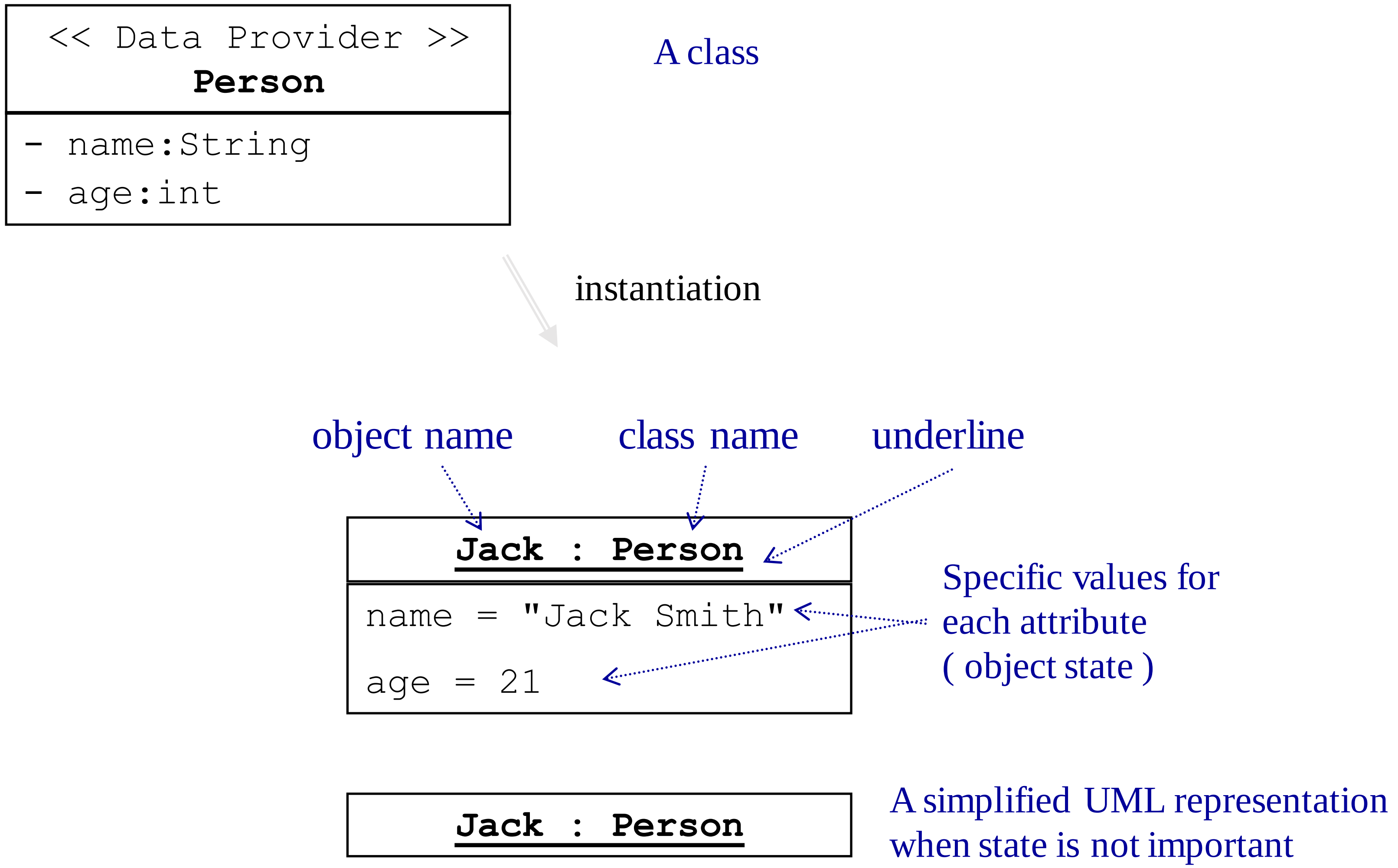
- : private attributes and methods

# : protected attributes and methods

{read only}: constant

___ (underline): static (class) attributes and methods

UML object symbol



Class Relationship

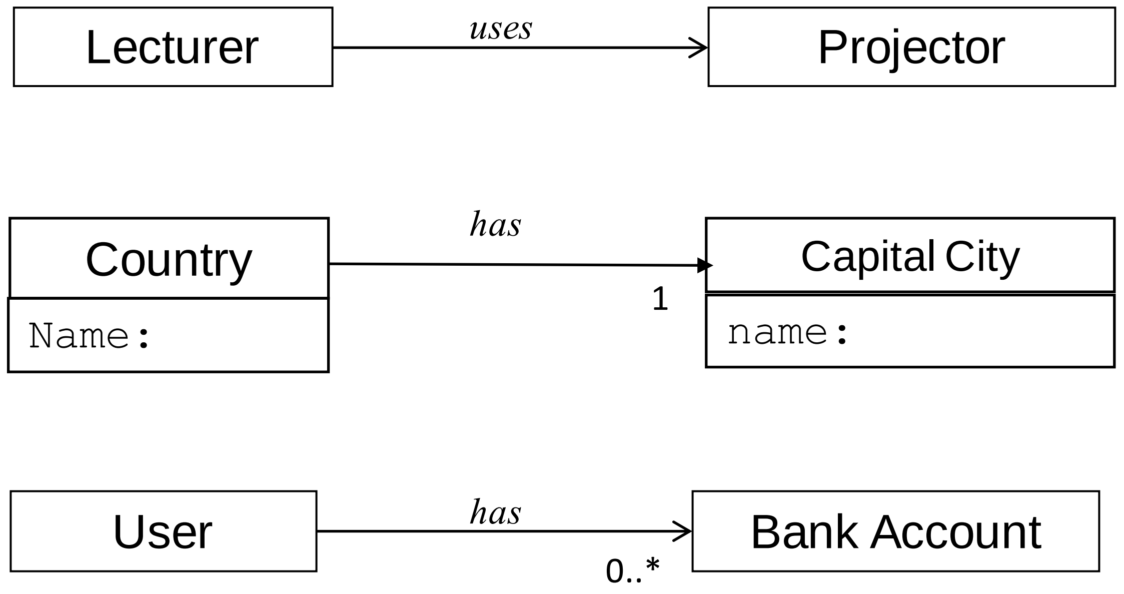
- Class symbols by themselves provide very limited information about your program architecture.

- A model of the system should reflect the relationship between classes

- OOD defines the following types of relationships between classes:
  - association
  - aggregation
  - composition

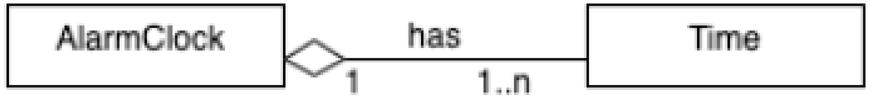
Association

- Indicates that one class uses another class
- Association can be described as “has” or “uses” type of relationship



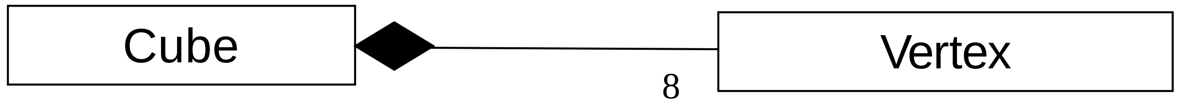
Aggregation

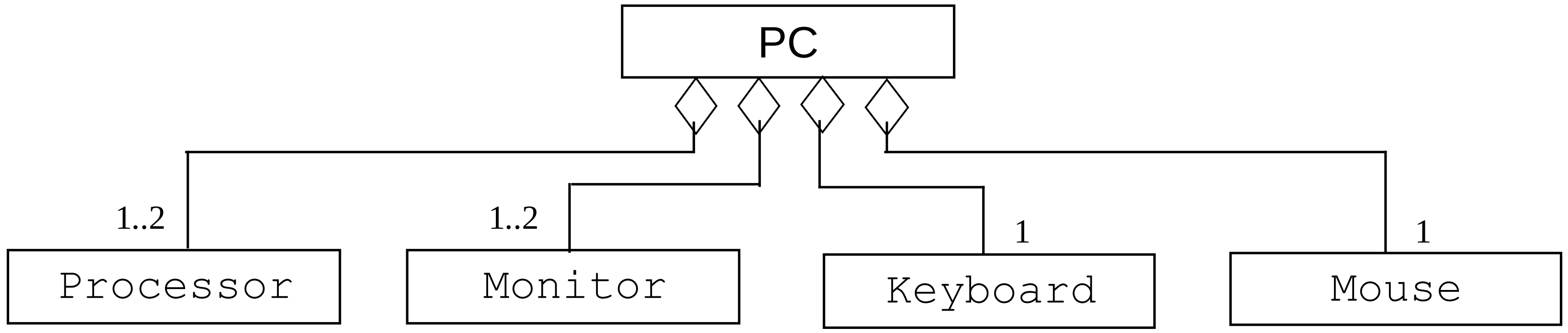
- A class can have references to objects of other classes as members.
- This is called aggregation and is sometimes referred to as a *has-a* relationship.
- The objects are fields of the class.
- Example: An AlarmClock object needs to know the current time and the time when it's supposed to sound its alarm, so it's reasonable to include two references to Time objects in an AlarmClock object.



Composition

- Composition is a special kind of association
- Composition reflects “contains” or “owns” type of relationship
- Components “live” inside the container with their lifespan synchronized with the container
- Deletion of the container destroys the component objects





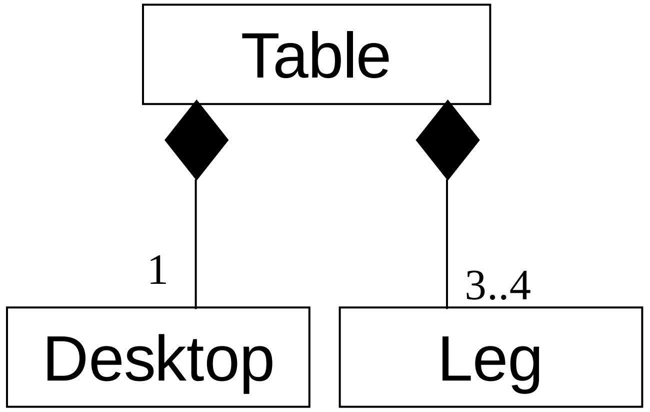
Quiz

- What is the most appropriate relationship between the following classes:

Table

Desktop

Legs



Defining classes

- The problem of defining an appropriate collection of classes and their relationship for your project is generally non-trivial
- Although there are some complicated theories which should help, practically you can mostly rely on your experience and intuition
- We are going to look at an made-up simple application scenario to show how to determine the classes and properties which seem appropriate

What makes a class model good ?

Two main objectives when constructing the class model:

1. build a system, quickly and cheaply, that *satisfies users'* needs
2. build a system that will be *easy to maintain* in the future

How are these objectives achieved?

1. "every piece of system behavior must be provided in a sensible way by the objects of the classes we choose"
2. "a good class model consists of classes which represent enduring classes of domain objects which don't depend on the particular functionality required today"

To do this, we...

- need some number of *design iterations* to develop a good class model
- focus on the *domain classes* first rather than classes arising out of implementation considerations

Identifying Classes

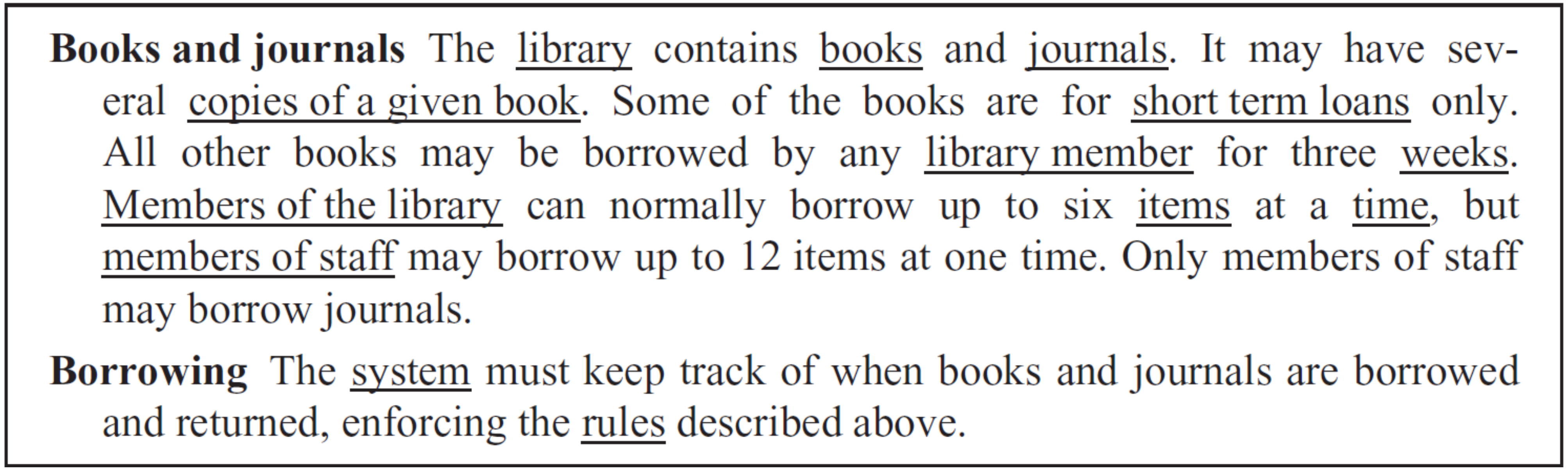
We use key domain abstractions

- *Domain* : application area we are working with, e.g. Library
- *Classes* are the principal abstraction of interest at this point
- How to identify the right classes?

Noun Identification Technique

This involves taking a coherent, concise statement of the requirements of the system and underlining its **<u>nouns</u>** and **<u>noun phrases</u>**; that is, identifying the words and phrases that **<u>denote things</u>**. This gives a list of **<u>candidate classes</u>**, which we can then whittle down and modify to get an **<u>initial class list</u>** for the system.

Identifying Classes



Books and journals The library contains books and journals. It may have several copies of a given book. Some of the books are for short term loans only. All other books may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.

Borrowing The system must keep track of when books and journals are borrowed and returned, enforcing the rules described above.

Noun Identification Technique

Nouns, noun phrases denote things, leads to candidate classes.

We discard:

- library (outside the scope)
- short term loan (an event)
- Member of the library (redundant, same as library member)
- week (time, not a thing)
- item (too vague)
- time (outside scope)
- system (meta-language, not part of the domain)
- rule (meta-language, not part of the domain)

Noun Identification Technique

First-cut list of possible classes:

- book
- journal
- copy (of book)
- library member
- member of staff

Relations between Classes

- Identify and name important real-world relationships or associations between classes
- to *clarify our understanding* of the domain, by describing our classes in terms of how they work together;
- record *multiplicities* of the associations

Relations between Classes

We can see that

- a copy *is a copy of* a book
- a library member *borrows*/*returns* a copy
- a member of staff *borrows*/*returns* a copy
- a member of staff *borrows*/*returns* a journal.

We record the information pictorially in the UML class model.

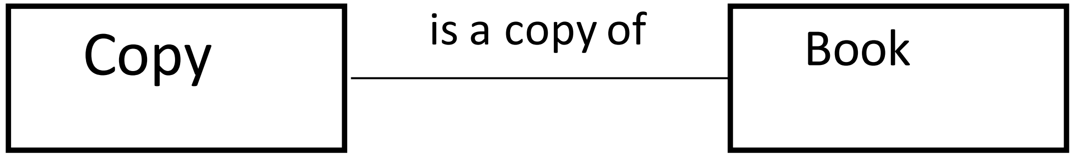
Associations

noun <--> class

verb <--> association

An instance of an association is called a *link* in UML.

A simple association is represented by a solid line between two classes



A simple association between classes

When are classes (A and B) associated?

- An object of class A sends a message to an object of class B;
- An object of class A creates an object of class B;
- An object of class A has an attribute whose values are objects of class B or collections of
- An object of class A receives a message with an object of class B as an argument.

A good class model...

does not distort the conceptual reality of the domain

but...

it also permits a sensible implementation of the required functionality

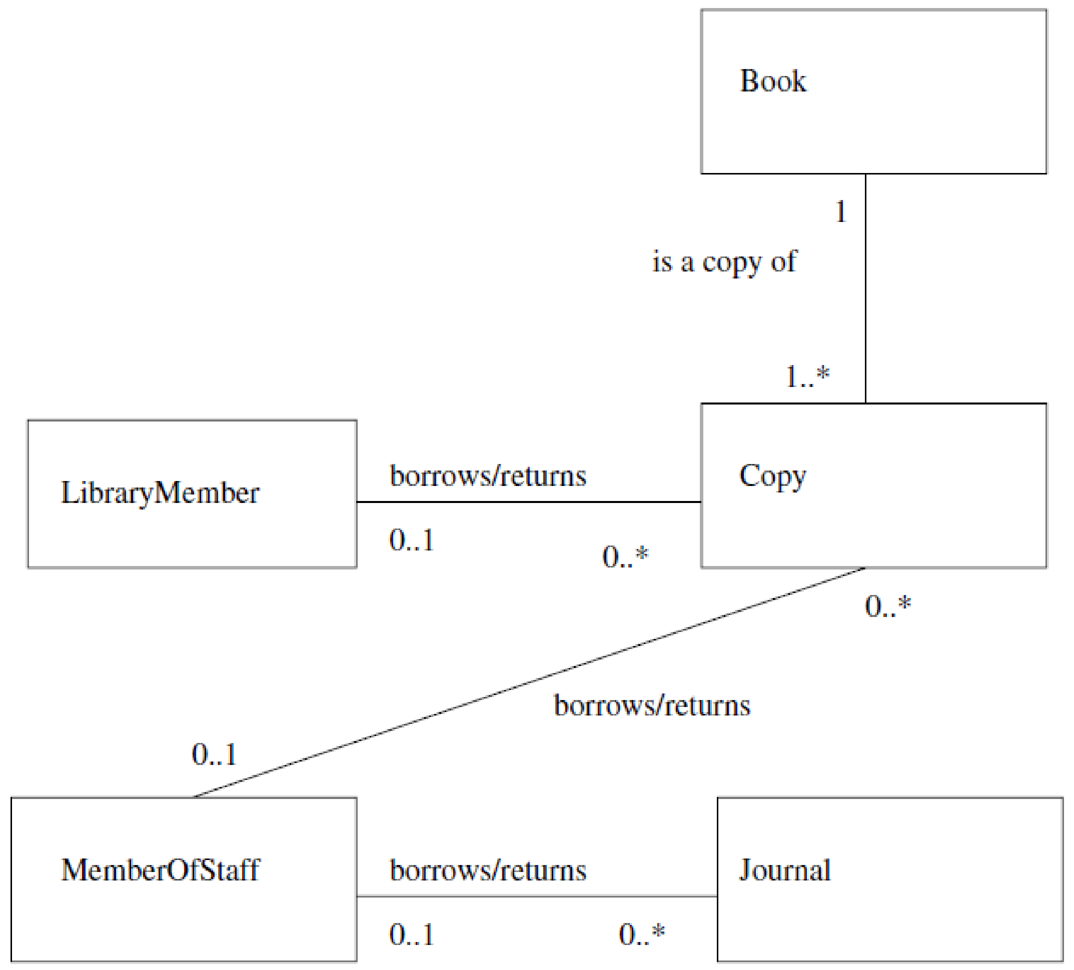
Multiplicities

We had showed a “1” at the Book end of the association “is a copy of” because every copy is associated with just one book.

Maybe any number of copies of a given book is needed, therefore, the multiplicity must be considered.

- An exact number – simple by writing it: 1 or 2,4,6,8
- A range of numbers – using two dots between a pair of numbers: 1..10
- An arbitrary, unspecified number - using * or n

Relations between Classes



Multiplicities

- 1: Each (each copy is a copy of just one book)
- 1..*: Maybe one or many (there may be on or many copies of a given book)
- 0..1: Maybe zero or one (a copy can not be borrowed by more than one person
- 0..*: Maybe zero or many (a person can borrow none or many copies of books)

Operations and Attributes

We had identified

- Classes
- Their relationships (association)

However, the system still cannot be proceeded without considering

- The state of objects of these classes
- The behaviour of objects of these classes

We need further to identify

- Attributes
- Methods

Attributes

The attributes of a class describe the data contained within an object of a class, they have:

1. Name
2. Type (either primitive types, integer, string etc, or classes not in the class diagram. If the type of an attribute is a class in the class diagram, it is better to record an association between the two classes)

Note

- check that enough data and behavior has been included to satisfy the requirements
- need to consider how objects will work (collaborate) together

Methods

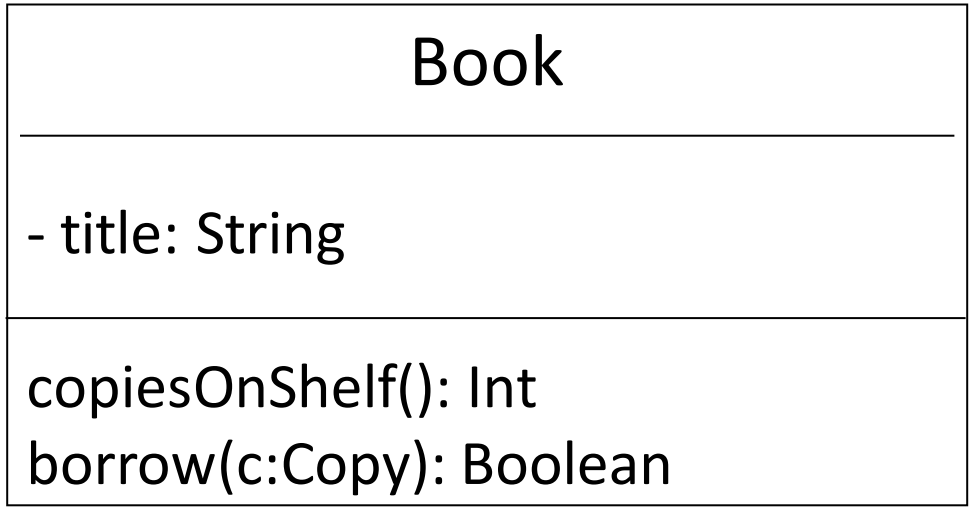
Important for a class as it defines ways in which

- an object processes data by itself
- an object interacts with others

A message received is a request to perform some methods, that is, execute some methods within a class

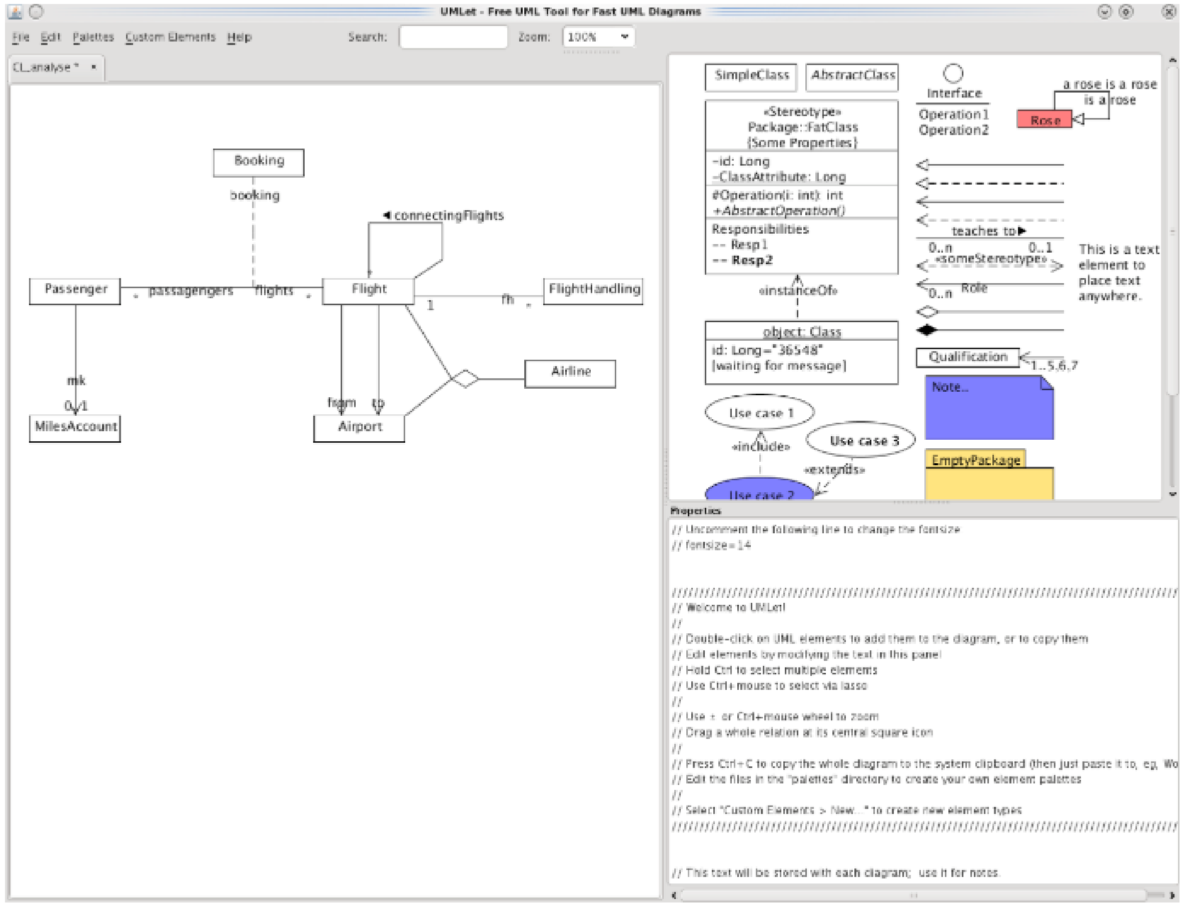
An method signature identifies:

- prefix
- name
- formal parameters (arguments, name & type)
- return value type



UML Design Tools

UMLet – Free UML Tool (online, download or Eclipse plugin)



```
// Uncomment the following line to change the fontsize
// fontsize=14


/////////////////////////////////////////////////////////////////////////////////////
// Welcome to UMLet!
//
// Double-click on UML elements to add them to the diagram, or to copy them
// Edit elements by modifying the text in this panel
// Hold Ctrl to select multiple elements
// Use Ctrl+mouse to select via lasso
//
// Use ± or Ctrl+mouse wheel to zoom
// Drag a whole relation at its central square icon
//
// Press Ctrl+C to copy the whole diagram to the system clipboard (then just paste it to, eg, Wo
// Edit the files in the "palettes" directory to create your own element palettes
//
// Select "Custom Elements > New..." to create new element types
/////////////////////////////////////////////////////////////////////////////////////


// This text will be stored with each diagram;  use it for notes.
```

Downloadable version: https://www.umlet.com/changes.htm
Online version: http://www.umlet.com/umletino/umletino.html

Suggested reading

Python 3 Object-Oriented Programming

- Preface
- Chapter 1: Object-Oriented Design

Python

- https://www.python.org/

UMLet

- https://www.umlet.com/

UML diagram

- https://www.softwaretestinghelp.com/uml-diagram-tutorial/