

Lecture 01

Object-Oriented Programming

Short review

Object-Oriented Concurrent Programming, 2019-2020

v1.6, 18-09-2019

Object-Oriented
Programming

OOP: The mechanics

Design by Contract

DbC: What For!

Assertions

Class Contract

DbC in Java

OO design techniques

Miguel Oliveira e Silva
DETI, Universidade de Aveiro

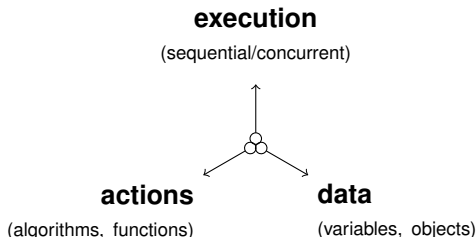
Object-Oriented Programming

- **Programming** can be defined as the act of creating, developing and implementing *solutions* to problems that can be *executed* by *computers*.
- These solutions are called **programs**.



- We can roughly separate two ways to solve problems and prescribe programs: one designated as **declarative** (functional, logical), and another **imperative** (procedural, object-oriented).
- Imperative programming differs from the declarative one because there is a *explicit* notion of a mutable *state*.
- An imperative program can be seen as a sequence of *commands* (or orders) which will change the state of the running system until the desired objectives are achieved.
- In imperative programming there are two dominant methodologies/languages:
 - **Procedural Programming** (Pascal, C);
 - **Object-Oriented Programming** (Java, C++, Eiffel).
- This course's goal is OO programming, hence we will (for the moment) stay focus only on imperative programming approaches.

- A program contains mechanisms that allow to **execute** certain **actions** on **data**.
- These are the **three forces** involved in computing:



- When you have to design the basic architecture of a program, the question arises as to whether it should be based on functions (actions) or data?
- It is in the answer to this question that essentially distinguishes procedural programming and object-oriented programming.

- It is rather easy to see that in theory, and if we ignore the fundamental aspect of abstraction, the two approaches are clearly dual to each other.
- However, in practice, the qualities of the programs developed by each of them are usually quite different.

The *Top-Down* procedural approach

- The *top-down* approach is based on the idea that a program is a stepwise refinement of the abstract function that defines it.
- The design and implementation of a program becomes a successive reduction in the functional abstraction until the entire program is expressed directly in the imperative instructions of the language or the use of functions of its libraries.
- It is a logical, systematic, well organized, and disciplined method.
- However, this approach suffers from the following problems:
 - It does not take into account the evolutionary nature of the software;
 - It is highly questionable whether a program should be defined by a function;
 - The priority given to functional abstraction has the consequence that data structures are often relegated to a minor secondary plan;
 - The *top-down* approach does not promote reuse;

Object-Oriented
Programming

OOP: The mechanics

Design by Contract

DbC: What For!

Assertions

Class Contract

DbC in Java

OO design techniques

- In general, functions are not a stable part of programs, hence to base the structure of programs on them can lead to radical changes only due to small changes in requirements (for example, if we move to a batch type interface for a more interactive one).
- **Compatibility**: It is difficult to combine actions if there is no compatibility in the data structures shared by them. Why not try combining data structures?
- **Reuse**: The *Bottom-Up* approach together with the data abstraction characteristic of OOP makes it easier to reuse and adapt existing modules.
- **Continuity**: Over time data structures, at least if viewed at an adequate level of abstraction, are in fact the stable aspects in a system.

- In structured procedural programming the construction of programs is done from “top-to-down”, identifying the functions of the programs and decomposing them step-by-step to a full implementation.
- In object-oriented programming, the construct is “bottom-to-up”, identifying and constructing the “objects” (or rather, the classes) involved in the problem, composing them to a full implementation.

To Think OO

The strategy to start to think OO is to organize the program structure around the **data** involved in the problem, making use of proper abstraction, and not primarily about what the program does.

Example problem: a paint alike to handle geometric shapes

Object-Oriented
Programming

Object-Oriented
Programming

OOP: The mechanics

Design by Contract

DbC: What For!

Assertions

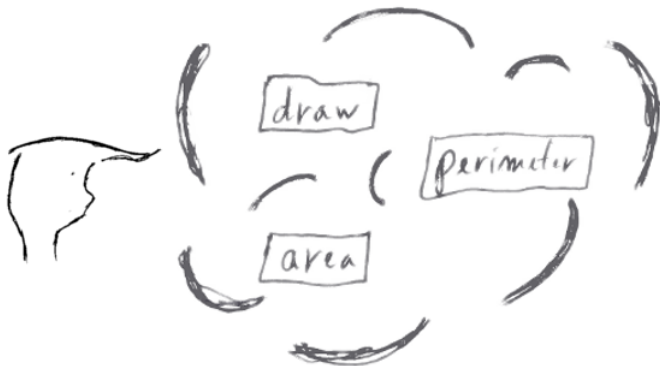
Class Contract

DbC in Java

OO design techniques



Example problem: Procedural approach



Example: a procedural approach to the perimeter of figures

```
double perimeter(Figure f) {  
    double result = 0;  
    switch(f.type) {  
        case RECTANGLE:  
            result = ...  
            break;  
        case CIRCLE:  
            result = ...  
            break;  
        case SQUARE:  
            result = ...  
            break;  
    }  
    return result;  
}
```

Example problem: Object-Oriented approach



Example: an OO approach to the perimeter of figures

```
class Figure {
    abstract double perimeter();
}

class Rectangle extends Figure {
    double perimeter() {
        return ...
    }
}

class Circle extends Figure {
    double perimeter() {
        return ...
    }
}

class Square extends Rectangle {
    ...
}
```

What are “Objects”?

- Object: data structure + operations
- Object: implementation of an Abstract Data Type (ADT)

Abstract Data Type

An Abstract Data Type is a data type defined only by the operations that are applicable to it and its properties (semantics)!

Object-Oriented Programming

Object Oriented Programming is the construction of software systems as structured collections of Abstract Data Types.

- Information hiding expresses the possibility of objects to explicitly define a set of exported services to external users (or hide a set of internal services).
- The object's ADT defines the services to export.
- Mutable object variables should never be exported.
- This is one of the essential mechanisms of OOP to achieve data abstraction.

- **Module view**: Construction of new objects by the difference from existing ones (extension).
- **Type view**: subtype polymorphism and dynamic binding (specialization).

- We can view objects as machines with an internal state (opaque) and to which we can apply commands and observe its state.
- It is very different to observe the state of a machine (query) and invoke an operation that can modify it (command).
- The first is associated with the mathematical notion of function and the second with the imperative notion of command or procedure.

Design by Contract

- Make the **semantics** (defined in the ADT) of the objects part of the **interface** and, if possible, to make it verifiable (statically or at runtime).
- This semantics is expressed by **assertions**: **invariant** of the object; **precondition**, and **postcondition** of each of its services.
- Here, the abstraction of data ceases to have a purely syntactic meaning (thus, only based on the goodwill and discipline of the programmers) to become a more complete, possibly verifiable, ADT.

- Maximize correction;
- Make the meaning of the code explicit within the program;
- Clear unambiguous distribution of responsibilities inside the program;
- Fault detection and tolerance.

- An **assertion** is a boolean expression, if possible, executable that expresses a property of the code.

```
void print(int[] list) {  
    for(int i = 0; i < list.length; i++) {  
        assert i < list.length;  
        out.print(" "+list[i]);  
    }  
    assert !(i < list.length);  
}  
  
boolean isPrime(int n) {  
    boolean result = true;  
    for(int i = 3; result && i*i <= n; i+=2) {  
        assert result && i*i <= n;  
        result = (n % i != 0);  
    }  
    return n == 2 || result;  
}
```

Preconditions and Postconditions

```
double sqrt(double x) {  
    // precondition:  
    assert x >= 0: "Negative x";  
  
    double result; (...)  
  
    // postcondition:  
    assert Math.abs(result*result-x) < MAX_ERROR;  
  
    return result;  
}
```

- How to guarantee certain properties in objects?
- For example, the absence of calendar date objects that do not match a valid date.
- Class invariant.

	Obligations	Benefits
Client	Satisfy the precondition of each service invoked.	Guarantee that both the class invariant the invoked service postcondition are verified when the service terminates.
Class	Ensure that the class invariant is verified at stable times. Ensure that, at the end of the execution of each of its services its postcondition is verified.	When a services is invoked, the precondition is verified.

The Java language does not adequately support Design by Contract. Some of its main shortcomings are the following:

- Does not distinguish between different types of assertions;
- There is no support for the definition of class invariants;
- Assertions are not part of the class interface;
- There is no inheritance of contracts;
- Documentation applications (`javadoc`) don't extract class contracts;
- The exception mechanism is not disciplined and was developed without considering DbC;
- We cannot activate and deactivate contracts by class.

OO design techniques

- Finding Objects:
 - External objects (books, people, figures, etc.);
 - Existing classes;
 - Adaptation by inheritance;
 - Define abstract classes by sharing properties (from ADT).
- The first big mistake:
promote a function to an object!
- Interface Techniques:
 - command query separation;
 - objects as state machines;
 - an interface, multiple implementations.
- The second big mistake:
confuse customers with heirs or vice versa!
 - **client**: has/uses
 - **heir**: is