



OOP PRINCIPLES REVISITED

INGEGNERIA DEL SOFTWARE

Università degli Studi di Padova

Dipartimento di Matematica

Corso di Laurea in Informatica

rcardin@math.unipd.it

WHAT IS IT?

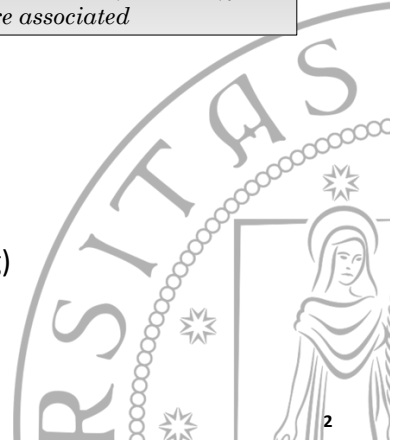
Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields; and code, in the form of procedures. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated

○ What is an object? And a class?

- Very easy to misunderstand

○ Three core principles

- Encapsulation (information hiding)
- Inheritance
- Polymorphism



WHAT IS IT?

○ The real problem is the definition of objects

- Messages (methods) and not data

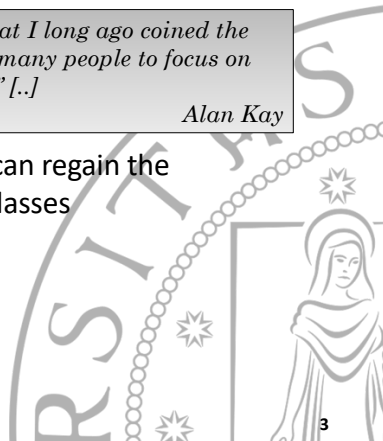
[...] it is not even about classes. I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea. The big idea is "messaging" [...]

Alan Kay

- Through the three principles, we can regain the correct definition of objects and classes

○ Based on extrinsic behaviour

- Naive objects hierarchies are evil



PROCEDURAL PROGRAMMING

○ Building block is represented by *the procedure*

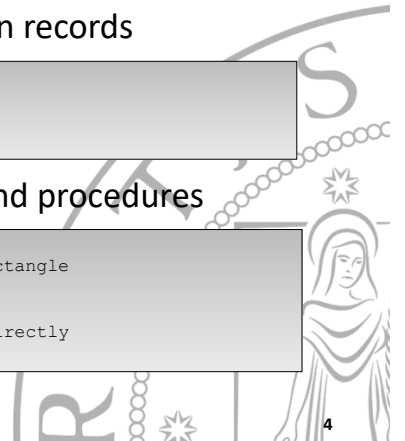
- Can have side effects

○ Data is primitive or structured in records

```
struct Rectangle {
  double height;
  double width;
};
```

○ No connection between data and procedures

```
double area(Rectangle r){
  // Code that computes the area of a rectangle
}
void scale(Rectangle r, double factor){
  // Code that changes the rectangle r directly
}
```



PROCEDURAL PROGRAMMING

- Procedures need the struct as input
 - Very verbose, hard to maintain, a lot of parameters

```
List<Double> scale(double height, double width, double factor)
```

- Lack of *information hiding*

- No restriction, no authorization process
- Testing is a hell

```
Rectangle r = new Rectangle(2.0, 4.0);  
r.height = 6.0  
printf(area(r)); // we expect 8.0, but a 24.0 is returned
```

OBJECT-ORIENTED PROGRAMMING

- Binding data with behaviours

The aim of Object-oriented programming is not modeling reality using abstract representations of its component, accidentally called "objects". OOP aims to organize behaviors and data together in structures, minimizing dependencies among them.

- The internal state is hidden from the outside

```
interface Shape {  
    double area();  
    Shape scale(double factor);  
}  
  
class Rectangle implements Shape {  
    private double height;  
    private double width;  
    /* Definition of functions declared in Shape interface */  
}
```

INFORMATION HIDING

- How to build a type using information hiding?

1. Find procedures sharing the same inputs
2. Get the minimum set of common inputs
 - Avoid tightly coupling
3. Create a structure using those inputs
 - Nope! Data is accessible from everywhere :(
4. Bind the structure with procedures, forming a type

- Clients must depend only on behaviour

- Hide data behind a private scope

- Use interfaces to hide implementations

INFORMATION HIDING

- Let's look at an example...

INHERITANCE

- Class (implementation)
 - Internal state and method implementation
- Type
 - The set of requests to which it can respond

Inheritance is a language feature that allows new objects to be defined from existing ones.

- Class inheritance (code reuse)
 - Reuse of object's implementation
- Interface inheritance (subtyping)
 - Reuse of object's behaviour

INHERITANCE

- Code reuse example

```
class AlgorithmThatReadFromCsvAndWriteOnMongo(filePath: String, mongoUri: String) {
  def read(): List[String] = { /* ... */ }
  def write(lines: List[String]): Unit = { /* ... */ }
}
class AlgorithmThatReadFromKafkaAndWriteOnMongo(broker: String, topic: String, mongoUri: String)
  extends AlgorithmThatReadFromCsvAndWriteOnMongo(null, mongoUri) {
  def read(): List[String] = { /* ... */ }
}
class AlgorithmThatReadFromKafkaAndWriteOnMongoAndLogs(brk: String, topic: String, mongoUri: String, logFile: String)
  extends AlgorithmThatReadFromKafkaAndWriteOnMongo(broker, topic, mongoUri) {
  def write(lines: List[String]): Unit = { /* ... */ }
}
```

INHERITANCE

- The banana, monkey, jungle problem

The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

Joe Armstrong

- Using a class adds a strong dependency also to parent classes
- Tight coupling
- One class, one responsibility
 - Single Responsibility Principle
 - Inheritance only from abstract types

INHERITANCE AND ENCAPSULATION

- Does class Inheritance break encapsulation?

- Classes expose two different interfaces
 - Subclasses can access internal state of base classes
 - Public and protected

- More and more clients for a class!!!

- Increasing of the dependency degree of a class
- The higher the dependency, the higher the coupling

- So, try to avoid class inheritance

SUBTYPING

Class inheritance defines an object's implementation in terms of another object's implementation. In short, it's a mechanism for code and representation sharing. In contrast, interface inheritance (or subtyping) describes when an object can be used in place of another.

- Inherit only from interfaces and abstract classes
 - Do not override methods
 - Do not hide operation of a parent class
- Loose coupling
 - Clients remain unaware of the specific type
 - Polymorphism depends on subtyping

WHEN TO USE CLASS INHERITANCE

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.
Liskov Substitution Principle

- Do not override pre- and post-condition of base class
 - Preconditions must be weaker, post conditions must be stronger than in the base class.
- Design by contract
 - Avoid redefinition of extrinsic public behaviour

COMPOSITION OVER INHERITANCE

- Black box reuse
 - Assembling functionalities into new features
 - No internal details

```
trait Reader {
  def read(): List[String]
}
trait Writer {
  def write(lines: List[String]): Unit
}
class CsvReader(filePath: String) extends Reader { /* ... */ }
class MongoWriter(mongoUri: String) extends Writer { /* ... */ }

class Migrator(reader: Reader, writers: List[Writer]) {
  val lines = reader.read()
  writers.foreach(_.write(lines))
}
```

CONCLUSIONS

- Define classes in terms of messages
- Never depend upon internal state
- Do not use class inheritance
- Favor composition over inheritance
- Design by contract
- ...
- Using inheritance and information hiding we built a procedure to define types in OOP

REFERENCES

- The Secret Life of Objects: Information Hiding
<http://rcardin.github.io/design/programming/op/fp/2018/06/13/the-secret-life-of-objects.html>
- The Secret Life of Objects: Inheritance
<http://rcardin.github.io/design/programming/op/fp/2018/07/27/the-secret-life-of-objects-part-2.html>

GITHUB REPOSITORY



<https://github.com/rcardin/swe>