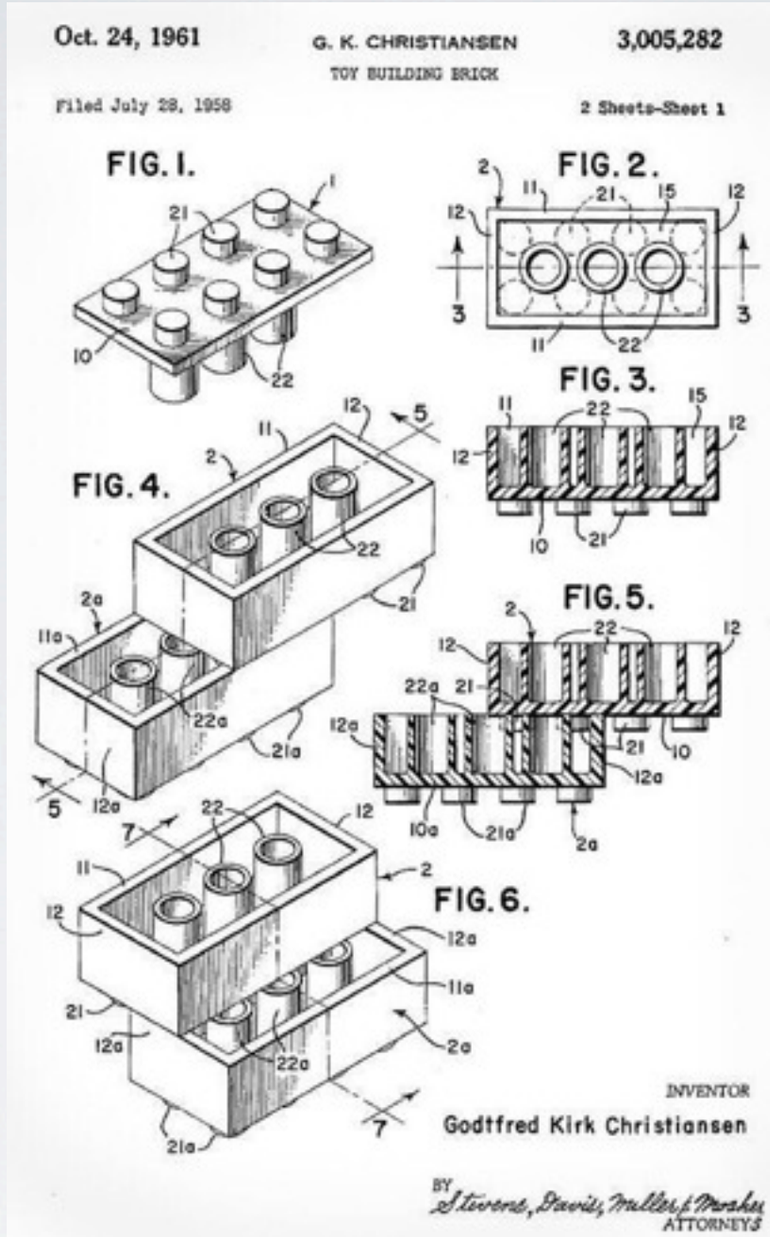
OBJECT ORIENTED PROGRAMMING

Introduction, concepts, design patterns, development C++/Java

Gilles BLANC — ESME Sudria september 2014



INTRODUCING OOP

INTRODUCTION

- *"Object-oriented programming is an exceptionally bad idea which could only have originated in California"* (Edsger W. Dijkstra *aka* God, 1930-2002)
- OUPS... http://harmful.cat-v.org/software/OO_programming/
- however:
  - object oriented programming (OOP) has enabled a new level/range of programming
  - OOP has solved much of classic coding problems (well, and created others, because one needs to think more about what she is doing...)
  - OOP is perfect match for graphic programming
  - most languages, reluctant at first, have become object oriented with new versions
  - lot of progress has been accomplished (or we hope so)

LiNACS
Embedded Linux project accelerator

WHY OBJECTS

- at the beginning was the spaghetti code...
  - no/very few functions/procedures
  - relying on goto (jumps to labels)
  - no abstraction, overlay to assembly
  - (not) planned obsolescence
  - nightmare
- soon (~45 years ago), a new way of programming appeared:
  - functions/procedures
- a function is "mathematical": inputs an output(s)
- a procedure can have side effects (printing to a screen, writing a file, etc.)
  - common mistake/language abuse & very few languages really make the difference (Ada, functional programming)
- goto declared illegal (except on system coding/low level programming), now punished by jail & whip
- but not sufficient enough for big programs...

LiNACS
Embedded Linux project accelerator

PROBLEMS TO SOLVE

- problems of:
  - complexity
  - maintenance
  - evolution
- all of this is related to the code organization
- ***the main problem is always <u>architecture</u>!!!!***
  (well, this and not programming like a pig, of course)

LiNACS
Embedded Linux project accelerator

Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it.

And to make matters worse: complexity sells better.

Dijkstra (1984) On the nature of Computing Science (EWD896)

IDEA OF OBJECT ORIENTED PROGRAMMING

- new paradigm (30 years ago): doing like nature!
  - an ***object*** is... an object (like a real object IRL)
  - a ***class*** is the idea of the object, an abstraction
  - the code defines the class, as something generic, it defines the **behavior**
  - data fields define the **state** of the object
  - so, an object is an instance of the class (created with an " ***instanciation***" by a **creator** method and then an affectation to a variable)
  - one class, one or several objects of the same class
- all the objects of a same class have the same characteristics than their class

LiNACS
Embedded Linux project accelerator

CHANGING PARADIGM

- In classic procedural language (such as C), procedures act on data
  - similar to functions in mathematics
  - data structures : packing of data according to a logical unit
- In object oriented programming, procedures are INSIDE data structures
  - a bit strange at first
  - actually, just think about real life: consider real life object, procedures become all actions, mutations, offered possibilities of the object
- Lot of theory associated with OOP, a bit scary, most often mind fuck. Design patterns to make it (relatively) cleaner
- Learning to **think object**

LiNACS
Embedded Linux project accelerator

CLASS AND OBJECTS

- objects of the same class...
  - share the same procedures ***(methods)***
  - share the same internal variables ***(attributes)***
    - but states of the attributes defer for each object, living independently (except if the attribute declared "static" or equivalent: it is then shared between all objects)
- modifications and reading of attributes SHOULD (MUST!!) be done via **mutator methods** (**accessors**/**getters** and **setters**)
- you can kill an object (***destruction***), never a class (but you can define a class inside a another class, *nested*)

LiNACS
Embedded Linux project accelerator

LET'S THINK ABOUT CAKES

- the mold class is a class
- the cake is an object
- recipe is the code (© pioupiou)
- baking is making an instance of object cake
- eating is deleting the object cake

(well, more or less, the mold is more like an object factory, actually, but pewpews tend to understand more this metaphor, above all when they're hungry)

LiNACS
Embedded Linux project accelerator

DECLARATION EXAMPLES

```
<myobject.h>


class MyObject
{
private:
   bool value;
public:
   MyObject() { value = true; }
};
```

```
<myobject.py>


class MyObject(object):
    def __init__(self):
        self.value = True
```

```
<myobject.java>


public class MyObject {
    private boolean value;
    public MyObject() { this.value = true; }
}
```

INSTANCIATION EXAMPLES

```
<myobject.cpp>

MyObject *myobj = new MyObject();
```

```
<myobject.py>

myobj = MyObject()
```

```
<otherobject.java>

MyObject myobj = MyObject();
```

BÜCHI
BÜCHI

ACCESSOR EXAMPLES

```
<myobject.h>

bool _value;

void value(bool val) {
    _value = val;
}

bool value() {
    return _value;
}
```

```
<myobject.py>

    def setValue (self, val):
        self.value = val
    def getValue(self):
        return self.value
```

```
<myobject.java>

boolean value;

void setValue(bool value) {
    this.value = value;
}

bool getValue() {
    return value;
}
```

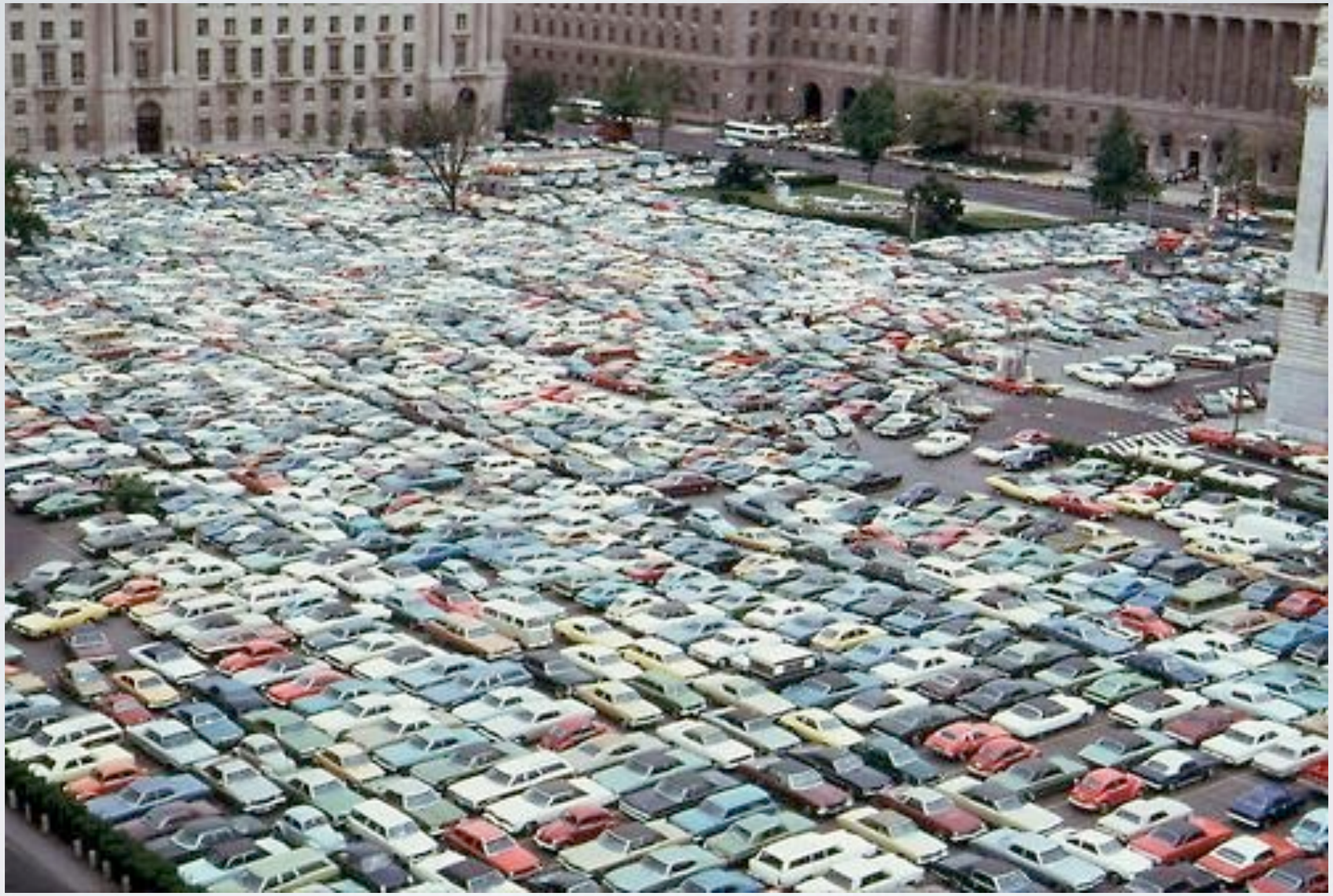
DESTRUCTION EXAMPLES

```
<myobject.cpp>


MyObject::~MyObject() { }

...
delete myobj; // myobj is a pointer
```

```
<myobject.py>


class MyObject(object):
    def __del__(self):
        pass
```

```
<myobject.java>

class MyObject {
     ~MyObject() { }
    ...
}

<otherobject.java>
delete myobj;  // normally called by garbage collector
```

```
// file car.h

class Car
{
private:
    unsigned char colorR, colorG, colorB;
    unsigned short speed;
    unsigned long x_pos, y_pos;
    float direction;
public:
    Car();
    void turn(float angle);
    unsigned long  get_x() { return x_pos; }
    unsigned long  get_y() { return y_pos; }
    void set_position(unsigned long x,
                      unsigned long y) {
        this->x_pos = x; this->y_pos = y;
    }
.........
};
```



```
for (int = 0; i < 1000; i++) {
    array[i] = new Car();
}
```

WHAT CAN WE DO WITH OBJECTS?

- organize code
  - each class has its own procedures
  - procedures are "stuck" with the class
  - so are attributes
  - prevent from doing huge tables with variables inside (different way of grouping, while thinking closer from nature)
- reuse code
  - inheritance system: a class can inherit procedures and attributes declaration from another class
  - encapsulation (objects containing other objects)
  - airlock organization preventing most of side effects due to code modification
- duplicate: several independent objects for the same class
- be modular, think by bricks and API for interactions

LiNACS
Embedded Linux project accelerator

JVC

"Which one of us is me?"

(QUICK) HISTORY

- Simula 67: evolution of Simula 1, first OO language ever
- Smalltalk (70's): Xerox innovations, huge impact (but forgotten)
- Common Lisp Object System (CLOS): adding objects to LISP in the 70's
- ObjectiveC (1983): smalltalk meets C, basis of MacOS (Cocoa), new life with iOS
- C++ (1983): most passional language ever...
- Object Pascal (1986), later Delphi (1995)
- Eiffel (1986): purely object-oriented language, considered as the best ever by many, never really used
- Python (1991), Ruby (1995): scripting languages appeared directly as object-oriented
- Ada 95: new version of Ada with objects
- Java (1995): influenced by C and Ada83
- C# (2000): trying to be a synthesis and learn from mistakes (C, Java, Eiffel influences)

LiNACS
Embedded Linux project accelerator

MORE OR LESS "OBJECTIFIED"

- some languages are just OOP extremists
  - Java may be the worst: each class is one file with the same name (so, one file per class), everything in a class (even if it is totally absurd)
  - Ada seems less intrusive while being rather strict
- some languages are more pragmatic (and permit very bad things too...): C++
- some languages are late (and rather badly) converted
  - Javascript: well, OOP aspect is not so obvious...
  - PHP, Perl, etc.: dirty languages...
- conversion of functional languages (like OCaml): does it really make sense?

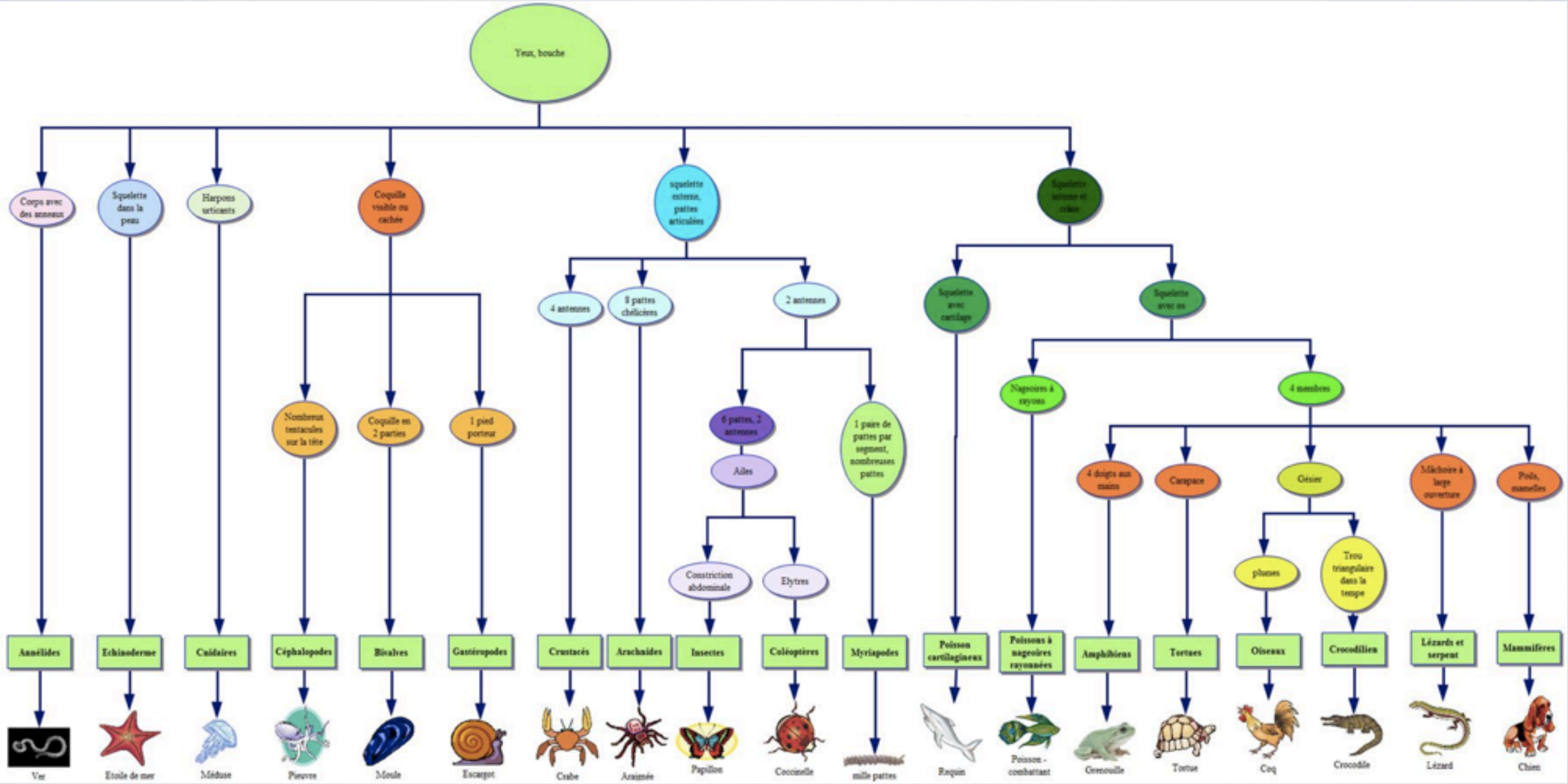
LiNACS
Embedded Linux project accelerator

INHERITANCE

- MAJOR paradigm of OOP
- a child class is derived from another class
- derived classes, subclasses, or child classes → base class, superclass, or parent class
- **relation: <A child> IS <B parent>**
- examples:
  - mammal is animal
  - dog is mammal, cat is mammal (so dogs and cats are mammal)
  - fish is animal, but fish is not mammal
  - catfish is not mammal (it is fish)

LiNACS
Embedded Linux project accelerator

INHERITANCE AS CLASSIFICATION/SPECIALIZATION



ANIMAL

Il a une tête et/ou une bouche et/ou des yeux

VERTEBRES : *Ils ont une colonne vertébrale.*

OSTEICHTYENS :

Ils ont un squelette osseux.

ACTINOPTERYGIENS :

« poissons » avec des nageoires à rayons

TETRAPODES

Ils ont 4 membres avec des doigts

AMPHIBIENS

Ils ont 4 doigts à la main.

MAMMIFERES

Ils ont des poils et/ou des mamelles.

TORTUES :

Elles ont une carapace sur le dos et le ventre

ARCHOSAURES

Ils ont un gésier.

OISEAUX

Ils ont des plumes.

CROCODILIENS

Ils ont un trou en forme de triangle sur l'os de la tempe

SQUAMATES : *Ils muent entièrement.*

CHONDRICHTYENS :

Poissons avec un squelette en cartilage

ARTHROPODES

Ils ont des pattes articulées et un squelette externe.

ANTENNATES : Ils ont des antennes.

MYRIAPODES :

ils ont de très nombreuses pattes

INSECTES

Ils ont 6 pattes et 2 antennes.

CRUSTACES :

Ils ont 4 antennes .

ARACHNIDES :

Ils ont 8 pattes.

ECHINODERMES

Ils ont de nombreux petits pieds à ventouses.

CNIDAIRES

Ils ont des tentacules urticants.

MOLLUSQUES :

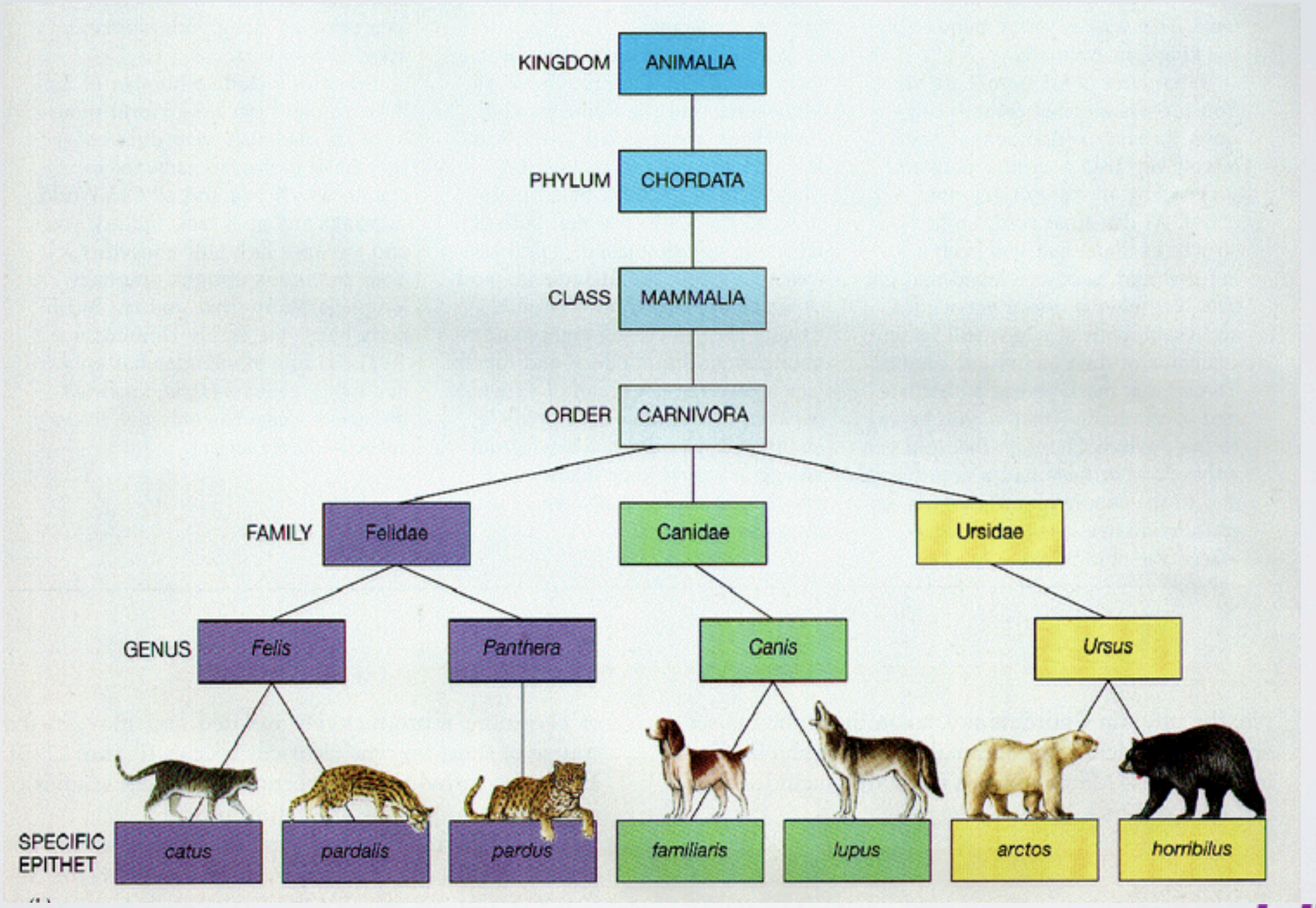
Ils ont une coquille visible ou non.

ANNELIDES

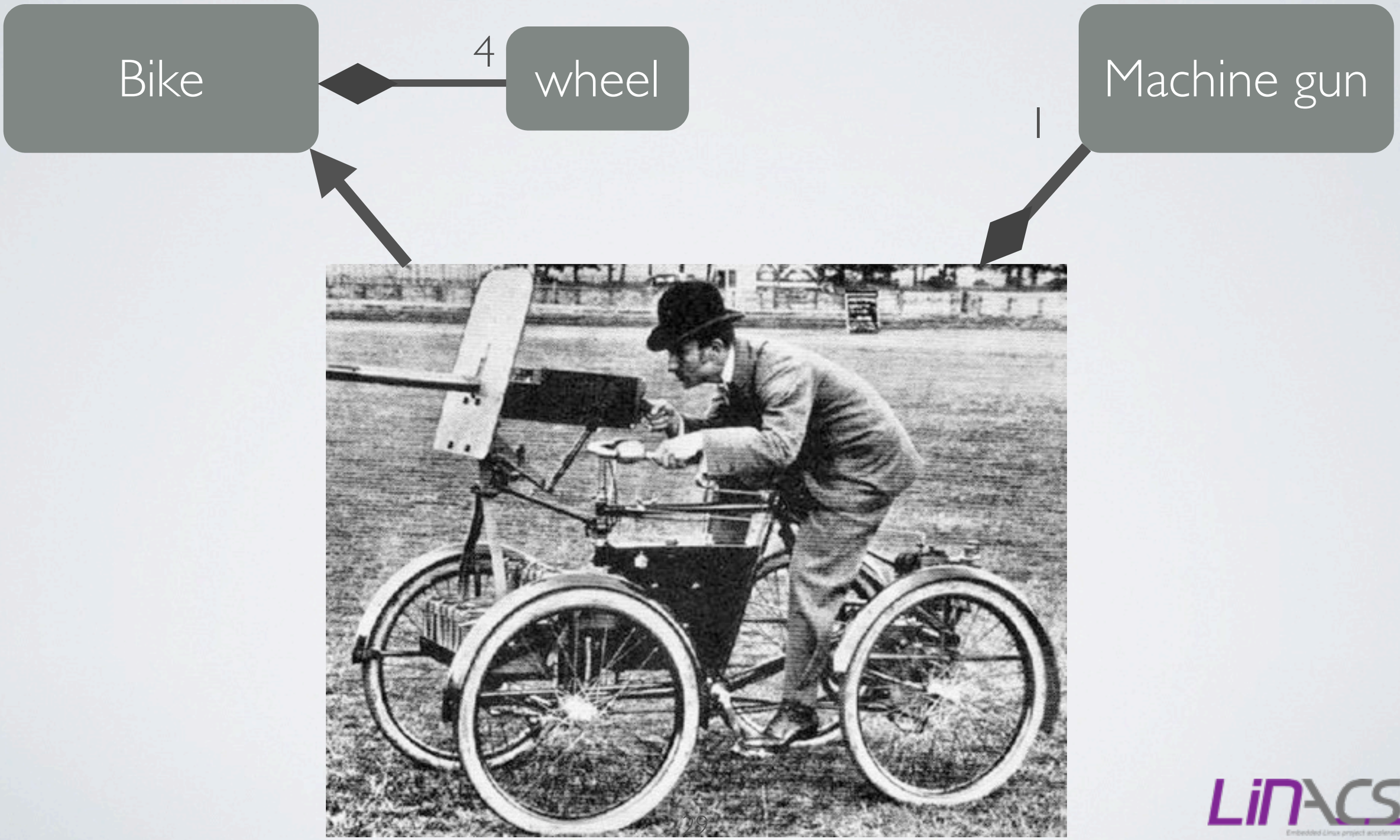
Ils ont un corps entouré d'anneaux.

CLASSIFICATION ACTUELLE DES ANIMAUX EN GROUPES EMBOITES

INHERITANCE & ABSTRACT CLASSES



UNDERSTOOD?



Vespa

Cannon

donkey
antitank
cannon

INHERITANCE EXAMPLES

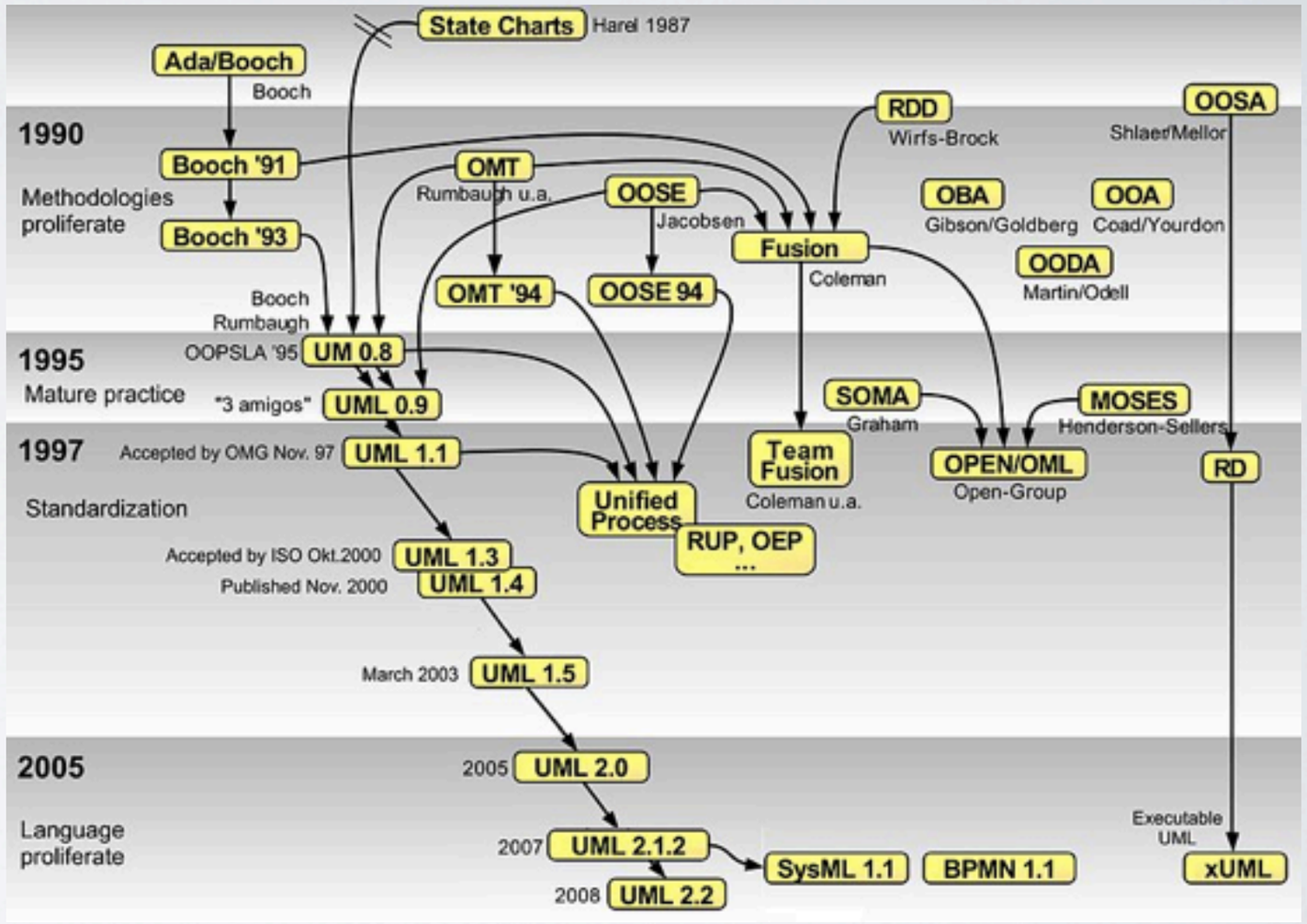
```
<myobject.h>

class shape
{ };

class circle: public shape
{ };
```

```
<myobject.py>

class shape:
    pass

class circle(object, shape):
    pass
```

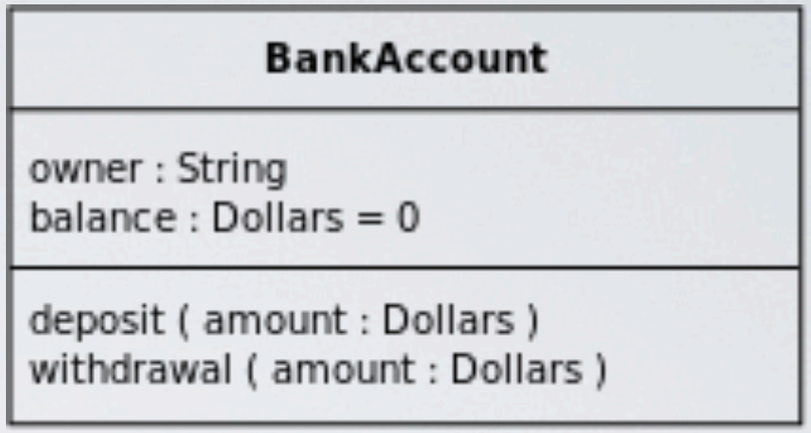
```
<myobject.java>

class shape {
}

class circle extends shape {
}
```

UML: CLASS DIAGRAM

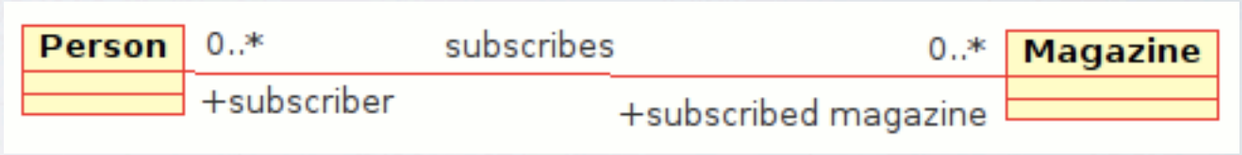
- need for notation of classes (spec & doc)
- long history
- classes (name, attributes, methods and visibilities), inheritance, aggregation/ composition



A class



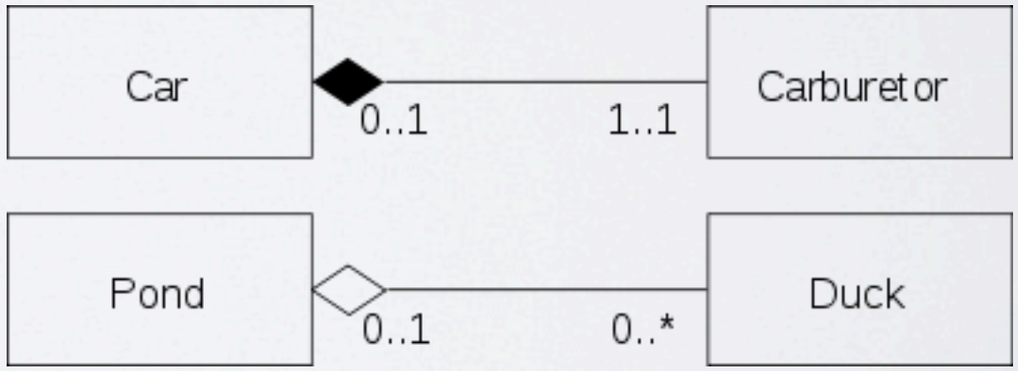
Association



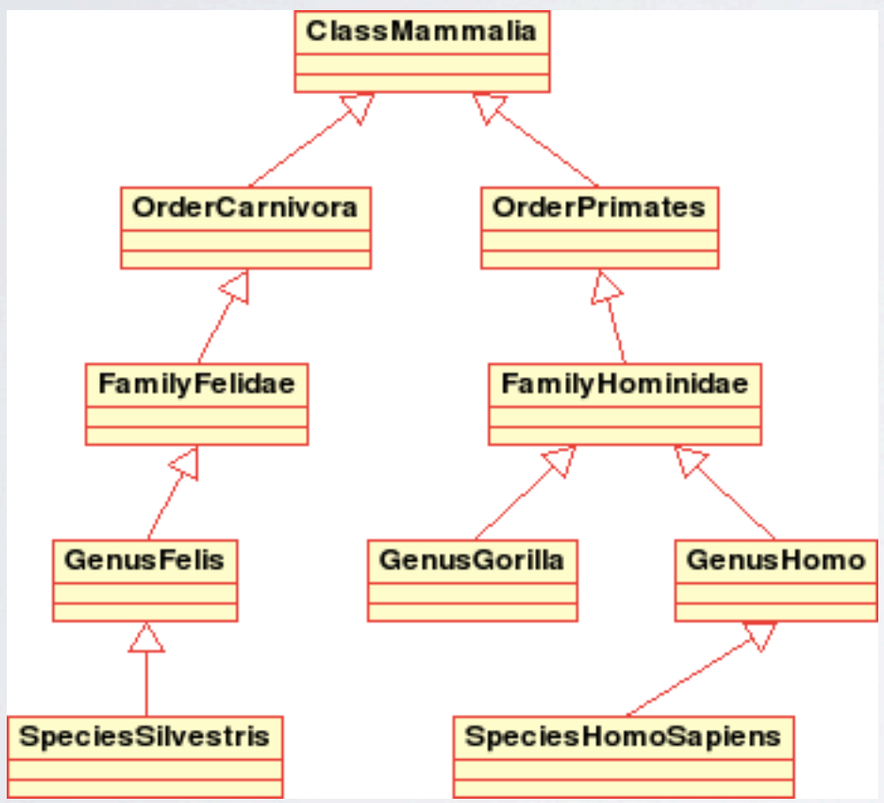
Aggregation ("has a")

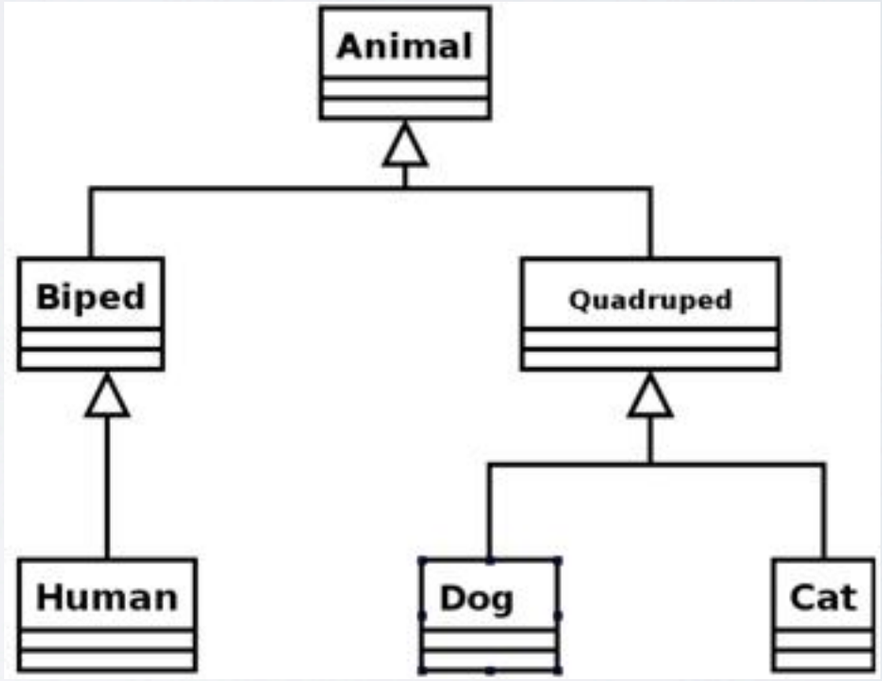
Composition
("has a" + strong life dependency)

(destroying a pond doesn't kill ducks)

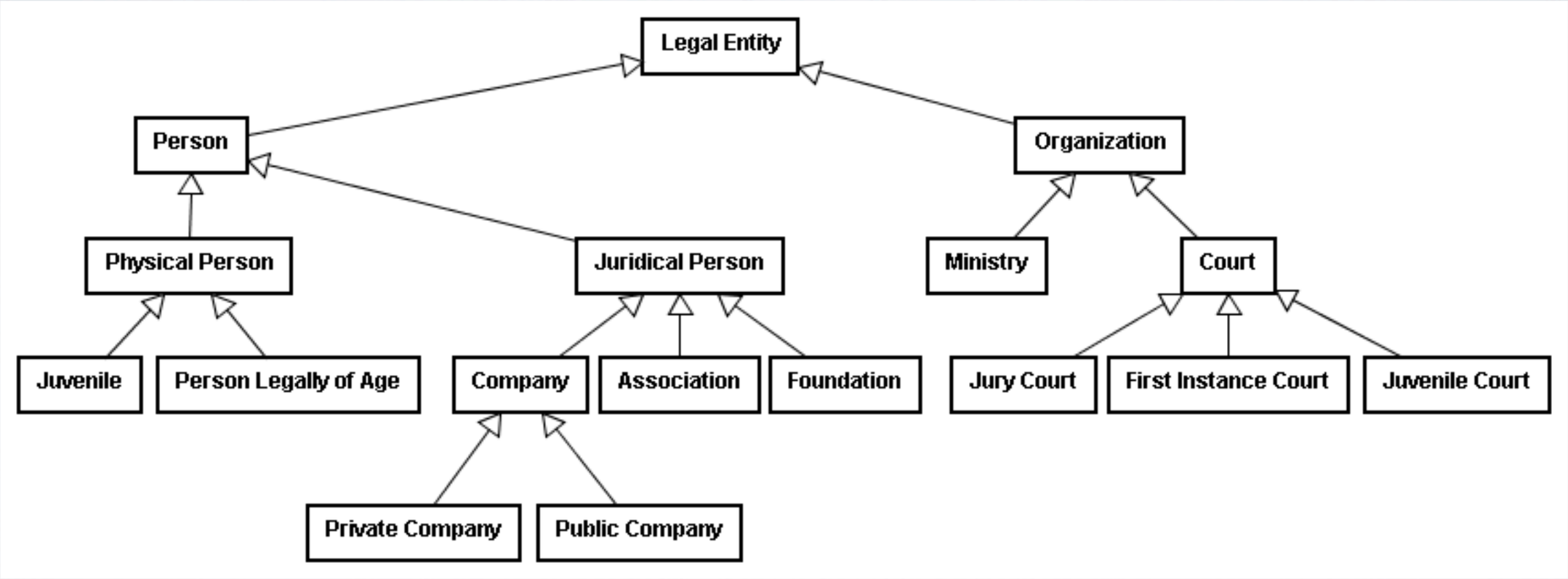


UML: ANIMALS

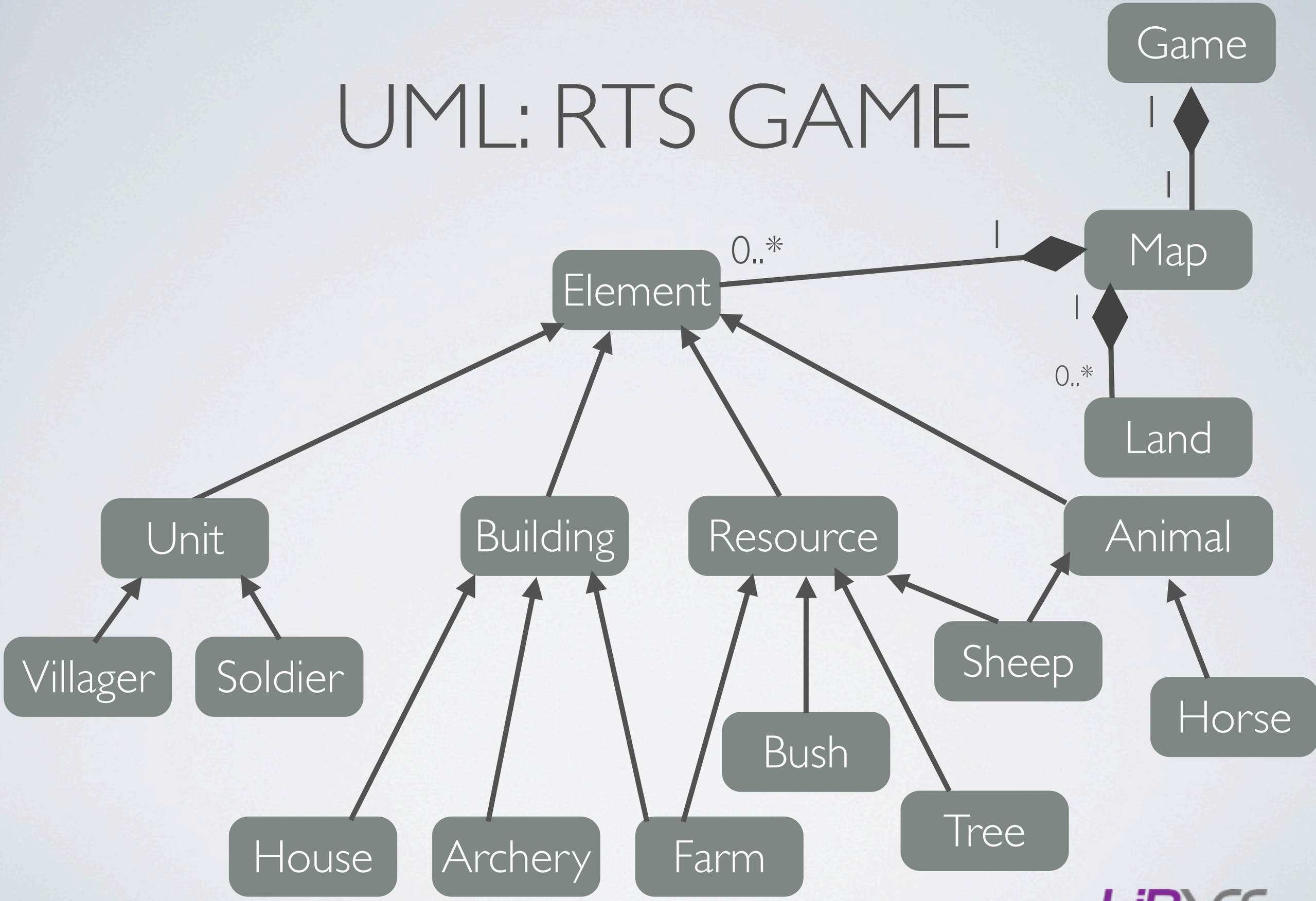

ClassMammalia
OrderCarnivora
OrderPrimates
FamilyFelidae
FamilyHominidae
GenusFelis
GenusGorilla
GenusHomo
SpeciesSilvestris
SpeciesHomoSapiens


Animal
Biped
Quadruped
Human
Dog
Cat

UML: LEGAL SYSTEM



UML: RTS GAME


Game
1
1
Map
1
0..*
Element
1
0..*
Land
Unit
Building
Resource
Animal
Villager
Soldier
Sheep
Horse
Bush
Tree
House
Archery
Farm

THINKING OBJECT

- modeling IS the key
- learn to divide the code in different objects with interactions
  - not too small, not too big
  - an object can rely on other objects: aggregation (the object bike uses objects wheels — if it has sense to divide like this, for example if the wheels have special behavior of their own, or their number can change)
- procedures in an object are acting like **inside** this object
  - side effects must always be internal
  - very different from a classic procedural language: you must change your point of view and think different©
- accessors are MANDATORY
  - even if the language is permissive, and you consider it stupid/useless
  - do not mess directly with object internal variables/attributes

LiNACS
Embedded Linux project accelerator

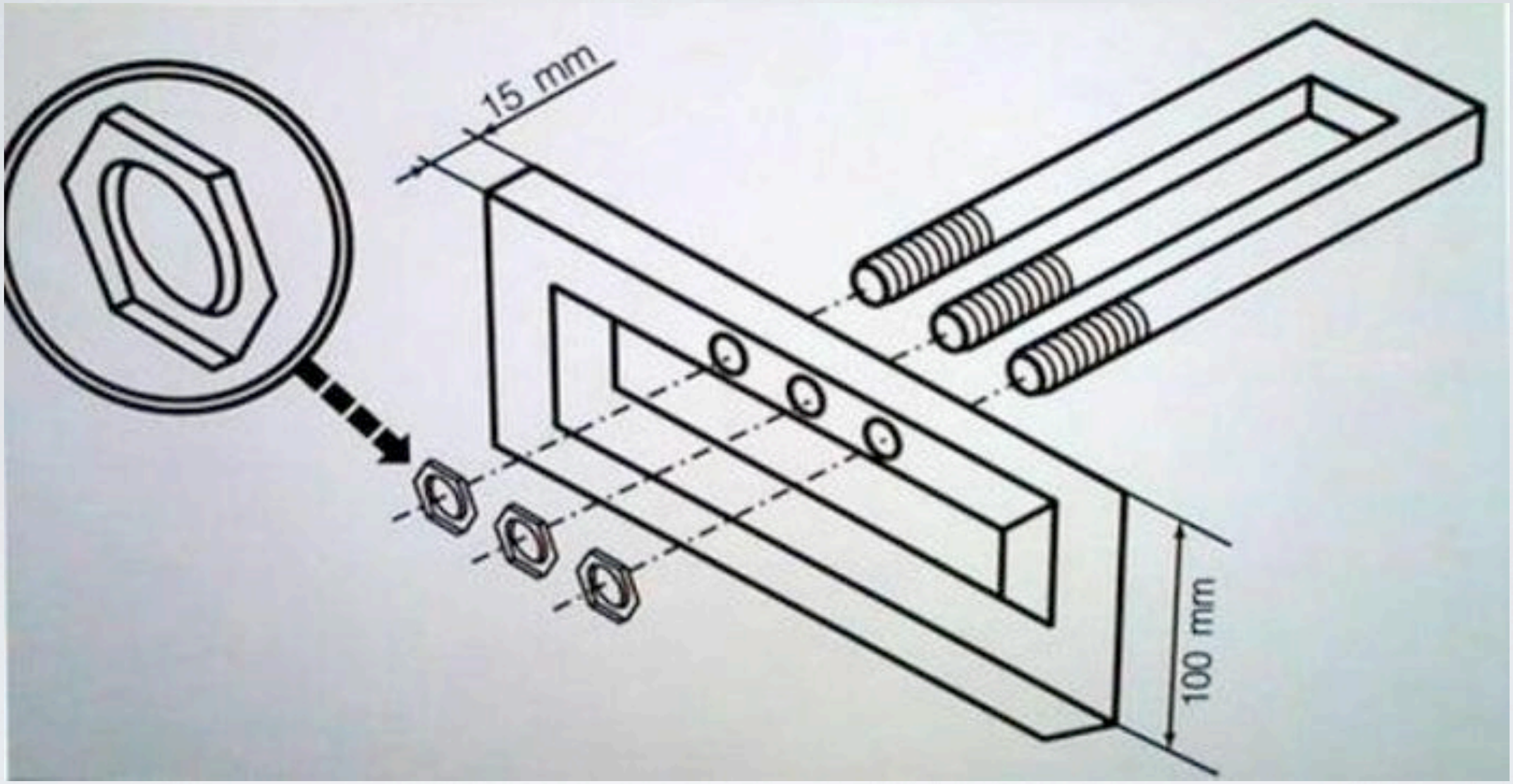
COMMON ERRORS

- **AN OBJECT IS NOT A SIMPLE COLLECTION OF PROCEDURES AND VARIABLES**
- creating an object is giving birth to an independent form of life, not a simple commodity
- a class/object is not a simple structure (not so evident with awful C++...)
- you must always remember to ***think object***
- messing between inheritance and aggregation
- C++ is NOT "C with objects"
  (unless you are an electronic engineer or an Chinese/Indian/assimilated, but please, save your soul!)

WHERE PROBLEMS COME

- is everything an object in life?
  - (very materialistic question, indeed)
  - for example, problem with algorithm, openGL, thread libraries...
    - sometimes, for the good of humanity, a class is more like a collection of procedures (which is bad, think again about Dijkstra saying)
    - when it is the case, we try to make the class abstract, to be unable to create objects, because it does not have any sense
    - hence the conception of "interfaces" in some OO languages (e.g. Java)
    - always remember: "with great power comes great responsibilities"
- ambiguities, several ways of expressing the same things: what is the right one?
- while coding, sometimes, you are stuck with an impossibility of doing what you want
  - often, problem of architecture/conception: consider refactoring the code!
  - but when it becomes "Mission: Impossible", use tricks: casting, interrogating nature of object (and specialized code), etc.

LiNACS
Embedded Linux project accelerator


15 mm
100 mm

HUGE & LETHAL MISTAKES

- inheritance
  - this is NOT a simple reuse of code with some kind of masking
  - you MUST ask yourself:
    if B inherits from A, is really B (a kind of) A?
  - only way not to mess with everything, killing architecture, and dying in huge pain (or worse: having to move to India and code like them)
- not using accessors/everything public and direct access
  - at first, seems quicker & easier, not boring with useless methods doing returns and affectations
  - then you modify something, someday, and everything explodes because of lack of airlock...

LinACS
Embedded Linux project accelerator

stadium

housing subdivision

WTF?!?

PARTICULARITIES

- what is called "paradigms"
- languages do not treat OOP the same way:
  - in Java, strict use of a file (delimitation)
  - in C++, all messy (with include files and code files, nightmare...)
  - in Python, you can change things inside an object or a class from the outside (for example changing a procedure): very powerful, very messy too ("use with extreme caution, risk of blowing")
- one has to understand how the creators of each language thought OOP and the best way to implement its code
- sometimes, you will curse the damn creator for having chosen ways against others, or having missed some important point (for you):
  - chose carefully your language for your project
  - use different languages
  - migrate from one language to another (well, tough one!)
  - implement your stuff in the language itself (if open source)
  - sacrifice a goat on your keyboard
  - create your own OOP language (for master jedis only)

LinACS
Embedded Linux project accelerator

PARADIGMS
&
MAIN DIFFERENCES BETWEEN LANGUAGES

KNOWING ONESELF

- common problem
- keywords depend on the language
  - more often "**this**" (pointer in C++)
  - "**self**" in Python
- knowing its family
  - in Java: super ("**super()**" calls the parent's class constructor)
  - in C++: problem with multi-inheritance, calling with parent class name as scope ("MyParentClass::method(argument);")

LiNACS
Embedded Linux project accelerator

POINTERS AND REFERENCES

- Pointer is a C speciality
  - a data points to another one
  - to be more precise: a register-sized data contains the address of a segment of a specific typed data
  - noted by a "*"
  - make the difference between the stack and the heap!! (VERY IMPORTANT) Instanciation in heap.
- Reference is the standard in other languages (actually, everything is (potentially?) a pointer, so no need to precise)
- In C++, you can either pass by pointer OR by reference ("&")
  - instanciation of an object as a pointer = "**new** object" / "**new** object(args)"
  - destruction of an object as a pointer = "**delete** object" / "**delete** object(args)"
  - same idea with tables of objects: "**new[]** objects" / "**delete[]** objects"
  - if you deal with objects as references, no need to manipulate pointers (and so to delete them when you are finished)

C++ CONST

- In C++, "const" keyword declares constant variable, or a method that don't modify the object data (e.g. read accessor/getter)
- No similar way in Java nor Python
- Verified during compilation
  - prevent to modify by error a data
  - very useful for optimization!!
  - can be mixed with references (either to return in a getter, or to return a temporary value of a function to a const container in the caller function)
    - `const MyObject foo() { MyObject myobj; return myobj; }`
      `void bar() { const MyObject &myobj = foo(); }`

LiNACS
Embedded Linux project accelerator

CONST & REFERENCES

```
class MyClass {
private:
    const char data;
// object stored inside this one (NOT a pointer!!)
    MyOtherClass insideObject;

public:
// using initialization list bypass const & shallow copy inside object
   MyClass(char data, MyOtherClass &ob) :
       data(data), insideObject(ob)
       {}
   char get_data const { return data; }
   const MyOtherClass &getinsideobj() {
        return insideObject;
   }  // this (returned) object will not be modifiable!
      // Only methods marked as ''const'' will be possible to be called
};
```

DEFAULT ARGUMENTS

- C++ & Python, but not in Java
- Method declaration with default arguments: if argument is not explicitly set with a value, the compiler takes the default value
- In C++, only in include file ".h" (declaration), not in ".cpp" code file
- Example: void function(bool bar, int val = 0);
- Problem with typing in C++
  - no strong typing, so it can mix up things, e.g. bool and int (yes, ugly...)
  - to prevent that, try to use only default values for last arguments of a method

CLASS VARIABLE

- Variable(s) common to all objects (contrary to "instance variable")
  - C++/Java: "static" keyword
  - Python: calling a variable without self (or with class name)
-
```
struct Request {
    static int count;
    int number;

    Request() {
        number = count; // modifies the instance variable "this->number"
        ++count; // modifies the class variable "Request::count"
    }
};

int Request::count = 0;
```

VISIBILITY

- For attributes and for methods
- Three kind of visibility:
  - **public**: everybody has access
  - **private**: only the class/object has access (inherited classes do not inherit those)
  - **protected**: like private but more liberal, as inherited classes of the class can access too
- NEVER put an attribute in public, use "private" and "protected"
- Do not apply to different objects of the same class, only for different classes
- In C++ and Java, not in Python nor nodeJS
- Keyword "**final**" (= no possible overriding/redefining in inherited classes)— in Java only
- In C++ (not in Java), when a child class inherit from a parent class, it can be specified if methods and arguments are let in the same state ("public" inheritance) or more protected ("private" inheritance)

LiNACS
Embedded Linux project accelerator

What happens when you put an attribute "public"

POLYMORPHISM

- Provision of a single interface to entities of different types
- A polymorphic type is a type whose operations can also be applied to values of some other type, or types
- Idea: use the same code with different types, being more abstract & general
- Different kinds:

1. ad hoc polymorphism: method overloading
2. subtyping (or simply "polymorphism" in OOP): method overring
3. parametric polymorphism aka generic programming: templates

LiNACS
Embedded Linux project accelerator

AD HOC POLYMORPHISM (1)

1 + 2 = 3
3.14 + 0.0015 = 3.1415
1 + 3.7 = 4.7
[1, 2, 3] + [4, 5, 6] = [1, 2, 3, 4, 5, 6]
[true, false] + [false, true] = [true, false, false, true]
"bab" + "oon" = "baboon"

```
int add(int x, int y)
{
    return x + y;
}

String add(String s, String t)
{
    return String(s).concat(t);
}
```

```
void main()
{
    System.out.println(add(1, 2));
    System.out.println(add('Hello, ', 'World!'));
}
```

IMPLEMENTATION IN LANGUAGES (1)

- Early binding:
  - calling of right function is determined by the compiler/interpreter
  - C++/Java/etc. ad hoc polymorphism is supported via **overloading**, based on argument types, not output types (very rare, must be strong typed at least)
- Late binding: supported by very few languages (Smalltalk)
- some languages only accept one function with the same name (for instance in Python, because argument types are not specified in function signature)
  - no user defined polymorphism allowed
  - or you have to determine by yourself what is the type (dynamic typing, some kind of late binding by hand)
  - using operator overloading: redefine operator '+' for example (possible in C++ and Python)

LiNACS
Embedded Linux project accelerator

OVERLOADING (1)

- function overloading of method overloading
- feature of some languages (e.g. C++, Java)
- allows to create several **methods** with the **same name**, which differ from each other in the type of the input (and/or the output of the function → rare)
- name masking and implicit type conversion problems to solve
  - sometimes very badly: C++ tends to mix booleans with integers for example
  - most commonly, output function is not enough to differentiate two methods (problem of not strongly typed languages like Java or C++)

LiNACS
Embedded Linux project accelerator

OPERATOR DEFINING/ OVERLOADING (1)

- e.g. redefining operator '+' between two classes, or access with '[ ]' like a table, etc.
- a little strange in Python (relying in introspection functionalities), but OK
- nothing in Java, very poor language...
- by design on C++ (limited set), very powerful
  - example:

```
Hydroplane::Hydroplane(Car &car, Plane &plane) {
  // .......
}

HydroPlane *Car::operator+(Plane &plane) {
     return new HydroPlane(*this, plane);
}
```

C++ OPERATOR OVERLOADING EXAMPLE (1)

```
Time operator+(const Time& lhs, const Time& rhs)
{
    Time temp = lhs;
    temp.seconds += rhs.seconds;
    temp.minutes += temp.seconds / 60;
    temp.seconds %= 60;
    temp.minutes += rhs.minutes;
    temp.hours += temp.minutes / 60;
    temp.minutes %= 60;
    temp.hours += rhs.hours;
    return temp;
}

bool Time::operator!() const
{
    return ((hours == 0) && (minutes == 0) && (seconds == 0));
}
```

CASTING, DETERMINING TYPE (1)

- Classic C casting
  - (ParentObject*)childObject
  - useful when wanting to consider a inherited class as its parent
- In C++:
  - dynamic_cast<ChildObject*>(object): return casting to child or NULL if no sense (at runtime)
  - const_cast<Object*>(object): transform const object pointer to non-const
  - static_cast<type>(object): similar to "(type)object"
- In Java: "instanceof" (infix keyword: "if (subobject instanceof Parentclass)")
- In Python: str.__class__ or "isinstance" to determine (manually at runtime) the class of an object

COERCION (1)

- very natural if language isn't strongly typed

```
double sqr(double x) {
    return x * x;
}

int main() {
    int i = 3;
    double res = sqr(i); // implicit cast/conversion => coercion
}
```

OVERRIDING (2)

- goal: to redefine a method of parent class inside the child class
- the new method overrides the functionalities and define new behavior when called
  - the method must have the same name, and same arguments of same types (different from overloading! But if overloading is supported, one can mix both)
  - the new procedure can call the mother class former procedure it overrides (for example to reuse functionalities and add its owns)
- with some languages (e.g. C++), one can call the class as its parent (with casting), and so the overridden method; this is not possible with Java, or if the method is marked as "virtual" in C++
- some languages allow to specify that a method must be overridden ("abstract" in Java, "pure" in C++)
- conversely, some languages allow to specify that a function cannot be overridden ("final" in Java, "sealed" in C#)

LiNACS
Embedded Linux project accelerator

POLYMORPHISM BY SUBTYPING EXAMPLES (2)

```
#include <iostream>
using namespace std;

class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0) : width(a), height(b) {}
      int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};

class Rectangle: public Shape {
   public:
      Rectangle(int a=0, int b=0) : Shape(a, b) { }
      int area () {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      Triangle( int a=0, int b=0):Shape(a, b) { }
      int area () {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};

// Main function for the program
int main( )
{
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   rec.area();
   shape = &rec;   // store the address of Rectangle
   shape->area();  // call rectangle area.

   tri.area();
   shape = &tri;  // store the address of Triangle
   shape->area();  // call triangle area.

   return 0;
}
```

POLYMORPHISM BY SUBTYPING EXAMPLES (2)

- previous code output, when executed:
```
Rectangle class: 70
Parent class area
Triangle class area: 25
Parent class
```
- When we call Rectangle or Triangle as Shape, Shape's "area()" method is used
- This can be prevented by declaring the "area()" method as virtual
- ```
  virtual int area() {
      cout << "Parent class area :" <<endl;
      return 0;
  }
  ```
- Note: as Rectangle and Triangle objects are Shapes, we could store them together in a list of Shapes. If we need then to determine which are their classes at runtime, we can use "dynamic_cast"/"instanceof"
  ```

ABSTRACT (2)

- An abstract class is a class that cannot be instanciated: it exists only for logical reason, for inheritance
- For instance, a "class Animal" doesn't exist by itself, only kinds of Animal exist: "class Fish", "class Elephant", "class Moth", "class Mouse"...
- A method (or an attribute, in Java) is not implemented in the class
  - in C++: marking a method as virtual pure, i.e. "**virtual** *return_type method_name*(*arguments*) **= 0**;"
  - in Java: marking class, method or attribute as "abstract" (simply by putting this keyword ahead of the declaration)
  - chess game example
    - "Piece" class cannot be instantiated: no sense to create a generic piece! So it is abstract.
      - however, King, Queen, Pawn, etc. classes inherits from Piece class
    - inside "Piece", the method "move()" cannot be defined (no sense)
      - only declared and let empty (in C++: "virtual void move() = 0;", in Java: "abstract move();")
      - however, each child of "Piece" must define its own way to move, with method overriding

LiNACS
Embedded Linux project accelerator

ABSTRACT EXAMPLES (2)

```
// C++ ''.h'' file

class ChessPiece
{
protected:
    unsigned char x, y;
public:
    ChessPiece() : x(0), y(0)
                 {}
    virtual void move() = 0;
};

class Pawn: public ChessPiece
{
public:
    void move() { y++; }
};
```

```
// Java file

class ChessPiece
{
    protected char x, y;
    ChessPiece() {
        x = 0; y = 0;
    }
    public abstract void move();
};

class Pawn extends ChessPiece
{
    public void move() { y++; }
};
```

PARAMETRIC POLYMORPHISM (3)

- Obvious in functional languages
- Can be achieved in C++ and Java with templates (less powerful)
- The goal is to define function or a data type generically, so that it can handle values identically without depending on their type
- `data List a = Nil | Cons a (List a)`

```
length :: List a -> Integer
length Nil          = 0
length (Cons x xs) = 1 + length xs

map :: (a -> b) -> List a -> List b
map f Nil          = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

TEMPLATES (3)

- parametric polymorphism
- very powerful in C++, copied in Java a few years ago (with more or less success), no real sense in scripting languages (not typed, or very generic containers, determination at runtime)
- goal: to specialize a class through an object with another class
- define generically, instantiate with typing (encapsulation)
  - for example, a container class vector of a certain class: the class has to know what type of objects will be stored inside (problem of space allocation)
  - very often used by libraries (e.g.: List<String>, List<int>, etc.)
- when you are gifted, very powerful way to specialize a generic class to others, that you can inherit
- you can template the template (beware of the headaches)

LiNACS
Embedded Linux project accelerator

CLASSIC C++ TEMPLATE USAGE (3)

```
class List<T> {
    class Node<T> {
        T elem;
        Node<T> next;
    }
    Node<T> head;
    int length() { ... }
}

List<B> map(Func<A,B> f, List<A> xs) {
    ...
}
```

EXAMPLE OF CUSTOM TEMPLATE USE (3)

- TimeTable: generic class of a time table (could be anything inside, abstract, table by days and hours of a yet unknown class type)
- ClassTimeTable: class inherits time table with template <Course> class (containing discipline & professor names)
- TeacherTimeTable: class inherits TimeTable with template <TeacherCourse> class (containing the name of the class to teach and course)

LiNACS
Embedded Linux project accelerator

C++ TEMPLATE EXAMPLE (3)

```
// .h file

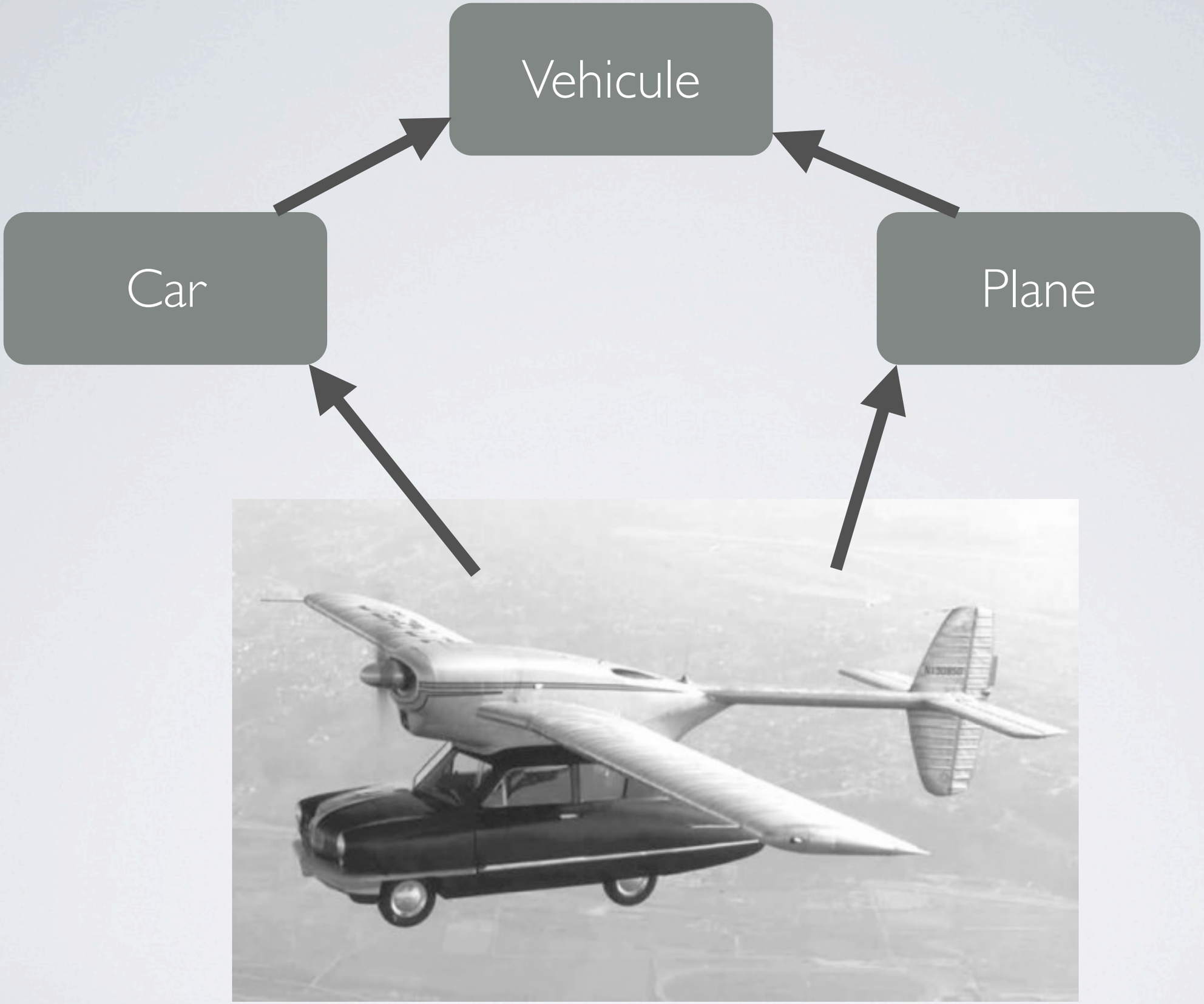
template <typename T>
class MyObject
{
    bool myfunction(int val);
}

// .c file
template<typename T>
bool MyObject<T>::myfunction(int val) {
    // do something
}
```

JAVA TEMPLATE EXAMPLE (3)

```
class Ref<Object Item>
{
    Item get(int i) {
        if (i<0)
            // legal BUT may generate an
            // exception if typing cast
            // doesn't mean anything
            return (Item)(Object)"no value";
        else
            return items[i];
    }
    Item items[];
}
```

GENERIC VS SPECIALIZED (3)

```
class Assoc<Object Key,Object Value>
{
    // méthode générique
    Value get(Key key)    { return hash.get(key); }

    // méthode spécialisée
    Value get(String key)    { return null; }

    Hash<Key,Value> hash=new Hash<Key,Value>();
}
```

Once compiled, as if we had implemented both following classes:

```
class Assoc<String,Object>
  {
    // méthode spécialisée
    Object get(String key)
    { return null; }

    Hash<String, Object> hash=
      new Hash<String, Object>();
  }
```

```
class Assoc<Object,Object>
{
  // méthode générique
  Object get(Object key)
  { return hash.get(key); }

  Hash<Object,Object> hash=
    new Hash<Object,Object>();
}
```

MULTIPLE INHERITANCE

- OK in C++, Python
- One class inherits several classes (not just one)
- Problem with "diamond" inheritance if parent classes are children of a same class and have both overridden a/several method(s)
  - brother and sister messing together + procedure overriding: which branch do you inherit?
  - You have to specify a "strong" branch and maybe to redefine some inherited methods
    - in C++, virtual inheritance of grandparent class + redefine
    - in Python: hierarchy goes from right to left
- KO in Java, no incest, not fun :-(
  (BUT: interfaces as workaround to this kind of problems, much less powerful, less potentially messy too)



How do you define "move()" method?

DIAMOND EXAMPLE IN C++

```
class Animal {
 public:
  virtual void eat();
};

class Mammal : public Animal {
 public:
  virtual void breathe();
};

class WingedAnimal : public Animal {
 public:
  virtual void flap();
};

// A bat is a winged mammal
class Bat : public Mammal, public
WingedAnimal {
};

Bat bat;
```

- Animal &a = bat;
- error: which Animal subobject should a Bat cast into, a Mammal::Animal or a WingedAnimal::Animal?
- Animal &mammal = static_cast<Mammal&> (b);
  Animal &winged = static_cast<WingedAnimal&> (b);
- static_cast<Mammal&>(bat).eat()
  static_cast<WingedAnimal&>(bat).eat()

DIAMOND SOLVING

```
class Animal {
 public:
  virtual void eat();
};
// Two classes virtually inheriting Animal:
class Mammal : public virtual Animal {
 public:
  virtual void breathe();
};

class WingedAnimal : public virtual Animal {
 public:
  virtual void flap();
};

// A bat is still a winged mammal
class Bat : public Mammal, public WingedAnimal {
};
```

OTHER CODE EXAMPLE

```
class Vehicle {
protected:
  int x, y, z;
public:
  virtual void move() = 0;
};
class Plane : public virtual Vehicle {
 public:
  virtual void move() { /* fly */ };
};
class Boat : public virtual Vehicle {
 public:
  virtual void move() { /* navigate */ };
};
class Hydroplane : public Plane, public Boat {
public:
  virtual void move() { if (z > 0) Plane::move(); else Boat::move(); }
};
```

PROTOCOL, MIXIN, TRAIT

- Sharing methods between classes with inheritance system without "is kind of" relationship
- Code reuse without multiple inheritance pathologies
- Protocol: methods between classes for their objects can communicate (see interfaces in Java, for example with a "compareTo()" method)
- Mixin: like an interface but with implemented methods (possible in Python by using multiple inheritance) and fields
- Trait: like a mixin, but without fields, only methods
  - class template used to associate state and/or behaviour to a compile-time entity, typically a data type or a constant, without modifying the existing entity
  - achieved by defining a primary class template, and creating explicit or partial specializations for the relevant types
  - in C++, widely used by standard library/STL (as a design pattern: no native implementation in language)

LiNACS
Embedded Linux project accelerator

PHP TRAIT EXAMPLE

```
trait Hello {
    public function sayHello() {
        echo 'Hello ';
    }
}

trait World {
    public function sayWorld() {
        echo ' World';
    }
}

class MyHelloWorld {
    use Hello, World;
    public function sayExclamationMark() {
        echo '!';
    }
}

$o = new MyHelloWorld();
$o->sayHello();
$o->sayWorld();
$o->sayExclamationMark();
```

PYTHON (KIND OF) MIXIN

- ```
  class Mixin1(object):
      def test(self):
          print "Mixin1"
  class Mixin2(object):
      def test(self):
          print "Mixin2"
  ```
- ```
  class MyClass(BaseClass, Mixin1, Mixin2):
      pass

  >>> obj = MyClass()
  >>> obj.test()
  Mixin1
  ```
- ```
  class MyClass(Mixin2, Mixin1, BaseClass):
      pass

  >>> obj = MyClass()
  >>> obj.test()
  Mixin2
  ```
  ```

JAVA INTERFACES

- No multi-inheritance in Java
- Interfaces are a way to declare methods a class must implement (some kind of "contract" you are waiting from an object)
- An interface is like an abstract class where all method are abstract: you only define signatures
- Can be implemented by one or several classes, which have to give an implementation for each announced method
- A class can implement several interfaces (allowing multiple inheritance)
- An interface has to constructor
- An interface has to field, except static
- An interface can be extended in one or other interfaces

LiNACS
Embedded Linux project accelerator

JAVA INTERFACE EXAMPLE

```
public interface I{
  public void A();
  public String B();
}

public interface I2{
  public void C();
  public String D();
}

public class X extends Y implements I, I2 {
  public void A(){
    //…
  }
  public String B(){
    //…
  }
  public void C(){
    //…
  }
  public String D(){
    //…
  }
}
```

EXCEPTIONS

- Very powerful way to handle errors
- When an error happens during runtime, the language allows to catch and treat them
- An exception is an object that can be filtered, with information inside
  - must be explicit in code by encapsulating sensible code (try/catch mechanism)
  - exception is transmitted from function to its callers until catch reached (if no catching, program fails)
  - be careful not to be too loose/catch to many thing: sometimes it is better to crash and correct the bug!
- OOP languages allow to create custom exception to throw
  - a interesting way to handle error cases (with much informations)
  - instead of returning a strange value from a function to be detected by a special routine of its caller, you can throw a special exception in your function and catch it in the caller

LiNACS
Embedded Linux project accelerator

JAVA EXCEPTION

- ```
  public static void main(String[] args) {
    int j = 20, i = 0;
    try {
      System.out.println(j / i);
    } catch (ArithmeticException e) {
      System.out.println("Division by zero ! " + e.getMessage());
    }
    System.out.println("Hey you!");
  }
  ```

- ```
  try {
      System.out.println(" =>" + (1/0));
    } catch (ClassCastException e) {
      e.printStackTrace();
    }
    finally{
      System.out.println("action done systematically");
    }
  ```

CUSTOM JAVA EXCEPTION

```
public class MyException extends Exception {
  public MyException(String message){
    super(message);
  }
}
void foo(int n) {
    if(n < 0)
        throw new MyException(n);
}
void bar() {
    try {
        foo(-1);
    }
    catch (MyException e | MyException2 e2) {
        System.out.println(e2.getMessage());
        e.printStackTrace();
    }
}
```

C++ EXCEPTION

```
try
{
  if (n == 1) throw 123; // throw an int
  if (n == -1) throw string("Fatal error"); // throw string
  if (n == 3) throw Object; // throwing an object
  int b = 42 * n;
  return b;  // this code is executed only if no throwing before
}
catch(int e)
{
}
catch(string const& e)
{
}
catch(Object const& e)
{
}
catch (...)  { }   // catches any error, with no filter
```

CUSTOM C++ EXCEPTION

```
#include <exception>
using namespace std;
 class MyError: public exception
{
public:
    Erreur(int num=0, string const& msg="", int level=0) throw()
      : num(num), msg(msg), level(level)  {}
     virtual const char* what() const throw() {
         return msg.c_str();
     }
     int getLevel() const throw() { return level; }
    virtual ~MyError() throw() {}
private:
    int num; string msg; int level;
};

int main(){
    try{
        throw MyException();
    }catch(MyException& e){
        cout<<e.what()<<endl;
    }
    return 0;
}
```

PYTHON EXCEPTION

```
print "beginning of code"

try:
    x = 'toto' + 1
except TypeError, e:
    print "Error of type: %s” % str(e)
except (RuntimeError, NameError):
    pass
except Exception, e:
    raise "Unknown exception: %s" % str(e)
finally:
    pass  #do cleaning here

print "end of code"
```

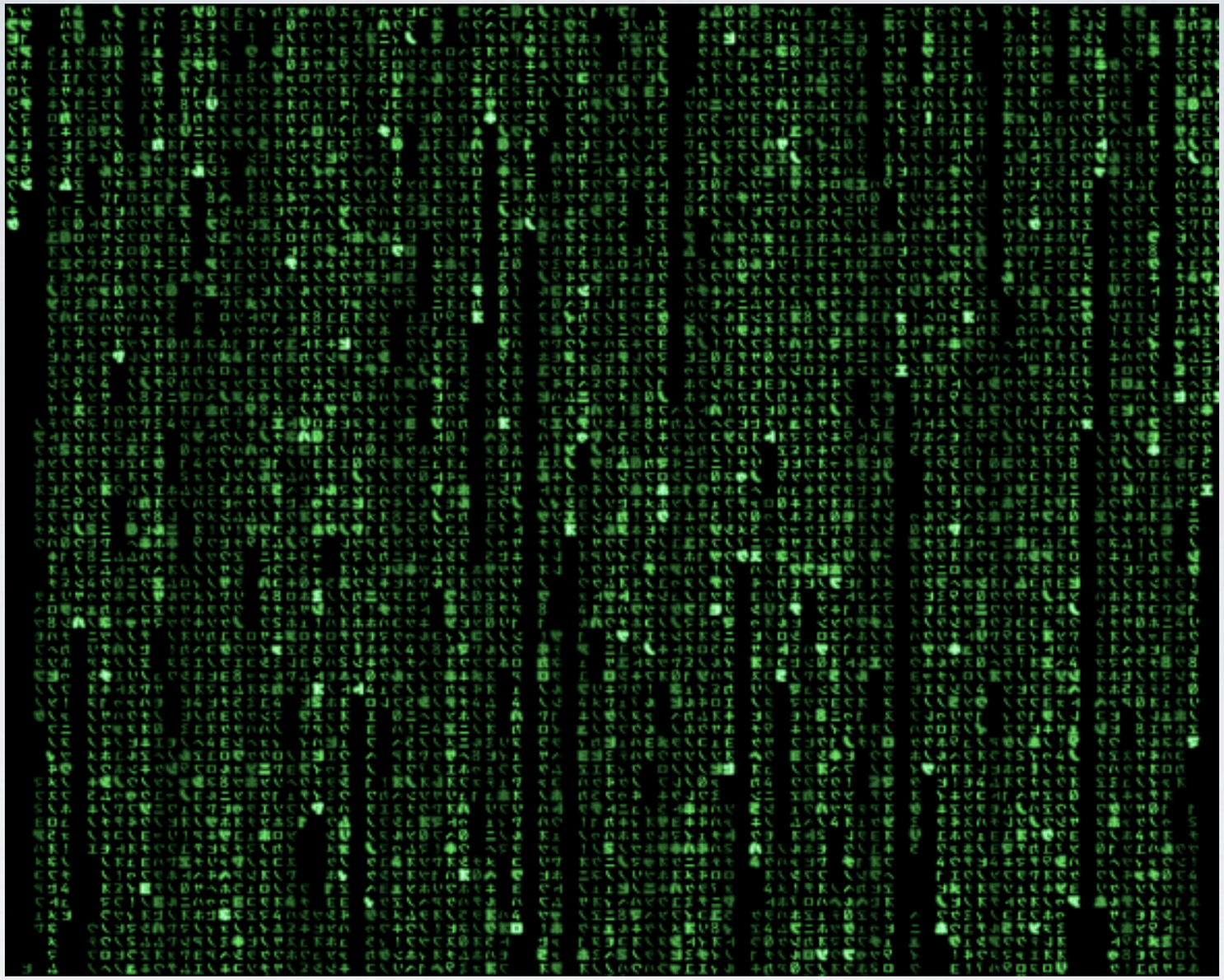
CUSTOM PYTHON EXCEPTION

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

INTROSPECTION / REFLECTION

- Seeing what's inside & potentially modify
  - for example, in Javascript, a object is seen as a table: a variable name is dynamic!
  - useful when treating data bases, more powerful than a simple hash table
- Implementation in languages varies much from one to another
  - very easy in Python (almost recommended)
  - more difficult in Java
    - to read: getClass(), getInterfaces(), getMethods()
    - to write: packages BCEL/Javasys, or by hand ("Class.forName()", "newInstance()", "getConstructor()")
  - impossible in C++
    - well, really difficult, only with procedure names, for backtracing, with black magic and only under Linux with gcc and debugging symbols
    - symbols are hashed, one may use c++filt binutil filter to demangle encoded C++ symbols

LiNACS Embedded Linux project accelerator



SHOW ME THE CODE!

C++

Test code online with: http://coliru.stacked-crooked.com/

CODE ORGANIZATION

- VERY SIMILAR TO C (you **must** master C first)
- Include files vs code files
  - h, hh: declaration and SOMETIMES code (when very short, often inline)
  - cpp, cc: code
  - causes a lot of nightmares. Keep calm, and include ".h" files a maximum in ".cpp", not in other ".h"!
  - protect your ".h" files with #define macros
  - declaration of data, structures/class (data and methods)
  - all methods must be implemented in include or code file, except if marked abstract ("=0;")
- Data in class is private by default. Data in struct is public by default
- In code files (".c")you can ONLY implement previously declared methods
  - you have to specify with a marker to which class you are refering (scope definition)
  -
```
void MyObject::myfunction() {
    // do something
}
```


LiNACS
Embedded Linux project accelerator

point.h

```
class Point {
private:
    int x, y;
public:
    Point(int x = 0, int y = 0) : x(x), y(y) {}
};
```

main.cpp

```
Point P1;   // static object

int main(int, char **)
{
  Point  p1;          // use constructor Point::Point(int, int)
                        // with default argument values
  Point  p2(10,20);     // Creates a new point with x=10 and y=20
  Point *p3;          // Declaration of pointer on a object of type
Point
  p3=new Point(20,30); // Instanciation of the dynamic object with explicit arguments
  p4=new Point;  // Instanciation of the dynamic object with default arguments

  delete p3;            // Never forget to free memory when dealing
with pointers!
  delete p4;
  return 0;
}
```

```
//  file <units.hh>

#ifndef UNITS_HH
#define UNITS_HH

#include "element.hh"
#include "sprite.hh"
#include "constant.hh"

#include <stdlib.h>
#include <vector>
#include <string>
#include <list>

class resource;
class building;
// code
class unit : public element {
public:
    unit() {}
    unit(int id_, int x_, int y_);
    virtual ~unit() { clear_actions(); };
    // ................
};
// ................

#endif
```

COMPILER ERROR OUTPUTS

- C++ compilers errors are quite ugly
- Sometimes very difficult to read and interpret
  - template method are almost impossible to read
  - "vtable" is the table of virtual functions to which a class with virtual functions points
  - macros can mess things
- Use colored output and/or a good IDE (Eclipse, VS, QtCreator)
- Try copy-paste to Google. Stackoverflow is very good!

LiNACS
Embedded Linux project accelerator

Les devises Shadok
ROUXEL
EN ESSAYANT CONTINUELLEMENT
ON FINIT PAR RÉUSSIR. DONC:
PLUS ÇA RATE, PLUS ON A
DE CHANCES QUE ÇA MARCHE.

LIBRARIES

- Some kind of default library: standard library (libstd: libstdc++.so.6 under Linux)
- "std::" (std::cout, std::list, etc.)
- #include <list>
  #include <vector>
  #include <algorithm>
- more or less the Standard Template Library (STL)

STL

- web:
  - Official web site: https://www.sgi.com/tech/stl/
  - Index of classes: https://www.sgi.com/tech/stl/stl_index.html
  - Example of class: https://www.sgi.com/tech/stl/List.html
- Old, not very sexy, direct cause of Java success
- Iterators everywhere, tricky, impossible to master correctly
- Quite poor, needs Boost librabry/tools as a complement to do (sometimes basic) interesting stuffs

LiNACS
Embedded Linux project accelerator

```
vector<int> v(3);          // Declare a vector of 3 elements.
    v[0] = 7;
    v[1] = v[0] + 3;
    v[2] = v[0] + v[1];        // v[0] == 7, v[1] == 10, v[2] == 17
```

```
for (std::list< std::pair<uint,uint> >::iterator i = movement.begin();
     i != movement.end(); i++)
    if (!game->get_map()->is_passable((*i).first / 32, (*i).second / 32))
        t->push_back(*i);

for (std::list< std::pair<uint,uint> >::iterator i = t->begin();
     i != t->end(); i++)
    movement.remove(*i);
```

Objazd
do Lędzin,
do Lędzin
6t
Imielin
6t
Lędziny
Tychy
44
934
S1
S1
S1
934
Mysłowice

```
enum e_sex { MALE, FEMALE, UNKNOWN };

class Dancer
{
protected:
    e_sex sex;
    char age;
    unsigned long x_pos, y_pos;
public:
    Dancer(e_sex sex, char age);
    void turn(rightside = false);
    unsigned long  get_x() { return x_pos; }
    unsigned long  get_y() { return y_pos; }
    void set_position(unsigned long x,
                      unsigned long y) {
        this->x = x; this->y = y;
    }
.........
};
```

```
class Ballerina: public Dancer
{
protected:
    bool wearing_tutu;
public:
    Ballerina() :
Dancer(), wearing_tutu(true) {}
}

class Corpsdeballet
{
private:
    std::list<Dancer*>;
public:
    Corpsdeballet(unsigned int number) {
        for (int = 0; i < 1000; i++) {
            Dancer << new Ballerina();
        }
    }
}
```

IMPLEMENTATION EXAMPLE (1/2)

- Configuration parser of files like:
  [GROUPPARAM]
  param=value
  param2=val2

```
std::pair <std::string, std::string> parseLine(const std::string &line) {
    std::string param, value;
    unsigned int n = line.find_first_of("=");
    if (n == line.npos) {
        if (line[0] == '[' && line[line.size() - 1] == ']')
            return std::pair <std::string, std::string>(line, "");
        throw "";
    }
    param = line.substr(0, n);
    boost::trim(param);
    if (param.empty()) throw "";
    value = line.substr(n + 1);
    boost::trim(value);
    if (value.empty()) throw "";
    return std::pair <std::string, std::string>(param, value);
}
```

IMPLEMENTATION EXAMPLE (2/2)

```
void ParseConfig(const char * const filename)
{
    std::ifstream input(filename);
    if (!input.is_open()) {
        std::cerr << "WARNING: conf file `" << filename << "' cannot be opened." << std::endl;
        return;
    }
    std::string line, groupparam; unsigned int numline = 0;
    while (std::getline(input, line)) {
        numline++;   boost::trim(line);
        //std::cout << line << std::endl;
        try {
            const std::pair <std::string, std::string> &paramvalue = parseLine(line);
            const std::string &param = paramvalue.first;
            const std::string &value = paramvalue.second;
            if (value.empty()) { groupparam = param; continue; }
            if (groupparam == "[GROUPPARAM]") {
                if (param == "param1") {
                    Config.val1 = boost::lexical_cast<unsigned int>(value);
                }
            }
        }
        catch (boost::bad_lexical_cast const& ) {
            std::cerr << "ERROR: Bad conf value at line " << numline
                      << ": `" << line << "'" << std::endl;
        }
        catch (...) {
            std::cerr << "ERROR: Unknown configuration line " << numline
                      << ": `" << line << "'" << std::endl;
        }
```

|||

FUNCTOR

```
class is_greater_than
{
public:
   is_greater_than (int n) : value(n) {}
   bool operator() (int element) const {
      return element > value;
   }
private:
   int value;
};

list<int> values;
// ... fill the list
if (find_if (values.begin(), values.end(),
             is_greater_than(5)) != values.end())
{
   // yes, it contains at least one number greater than 5
}
```

QT5

- Qt more and more preferred:
  - cleaner
  - more documentation
  - more optimized & stable
  - better paradigm
  - lot more of functions
  - able to deal with graphics (first usage!)
  - commercial support (Digia)
  - portable: Linux, MacOS, Windows, Blackberry, Android....
- http://qt.digia.com/
- http://qt-project.org/doc/ http://qt-project.org/doc/qt-5/classes.html

LiNACS
Embedded Linux project accelerator

QT CODE EXAMPLE

```
bool MainWindow::saveAs()
{
    QFileDialog dialog(this);
    dialog.setWindowModality(Qt::WindowModal);
    dialog.setAcceptMode(QFileDialog::AcceptSave);
    dialog.exec();
    QStringList files = dialog.selectedFiles();

    if (files.isEmpty())
        return false;

    return saveFile(files.at(0));
}
```

C++ TRICKS

- **namespace**: declare a global scope (often used with "std" when using STL), no need to use "ObjectName::" (beware of ambiguous names)
  - "**namespace** std **{** ..... **}**" for a local scope
  - "**using namespace** std;" for a global scope
- **virtual**: force reimplementation of marked method in each child class, no casting to parent to call the method
- **template & forced instanciation**: when you use a template method of the parent class inside a code file which is not corresponding to its include
  in ".h": `class MyChildObject : public MyParentObject<templateType> {`
  `bool myfunction(int val);`
  `}`
  in ".cpp": `template bool MyParentObject<templateType>::myfunction(int val);`
- **pointer to a method** (like function pointer in C)
  `bool (MyClass::*myfunction) (int, char, bool)`
- **friend**: a class marked as friend in another one can have its private/protected data bypassed; VERY DANGEROUS; useful only when dealing with nested classes, to prevent coding of too many (unwanted) accessors

C++ NINJA TRICKS

- **mutable**: when in a const method (e.g. a getter), you cannot modify object internal parameters. What if you want to manage a cache for optimization? Declare the cache parameter as mutable, this gives you the right to modify and bypass any "const" declaration (VERY DANGEROUS, USE WITH CAUTION)
- **auto**: since C++11 (2011)
  - type inference: when declaring a variable of type auto with a mandatory instanciation, the compiler guess its type
  - useful with templates
  - also: `int i; decltype(i) j = 5;`

LinACS Embedded Linux project accelerator

- ```
  #include <iostream>
  #include <cmath>
  #include <typeinfo>

  template<class T, class U>
  auto add(T t, U u) -> decltype(t + u) // the return type is the type of operator+(T,U) {
      return t + u;
  }

  auto get_fun(int arg)->double(*)(double) // same as double (*get_fun(int))(double) {
      switch (arg) {
          case 1: return std::fabs;
          case 2: return std::sin;
          default: return std::cos;
      }
  }

  int main()
  {
      auto a = 1 + 2;
      std::cout << "type of a: " << typeid(a).name() << '\n';
      auto b = add(1, 1.2);
      std::cout << "type of b: " << typeid(b).name() << '\n';
      //auto int c; //compile-time error
      auto d = {1, 2};
      std::cout << "type of d: " << typeid(d).name() << '\n';

      auto my_lambda = [](int x) { return x + 3; };
      std::cout << "my_lambda: " << my_lambda(5) << '\n';

      auto my_fun = get_fun(2);
      std::cout << "type of my_fun: " << typeid(my_fun).name() << '\n';
      std::cout << "my_fun: " << my_fun(3) << '\n';
  }
  ```

- http://en.cppreference.com/w/cpp/language/auto
  ```

OTHER EXAMPLE

- ```
  #include <iostream>
  #include <string>
  #include <vector>

  template<typename T>
  std::ostream& operator<<(std::ostream& os, const std::vector<T>& vec)
  {
      for (auto& el : vec) {
          os << el << ' ';
      }
      return os;
  }

  int main()
  {
      std::vector<std::string> words = {
          "Hello ", "from", "GCC", __VERSION__, "!"
      };
      std::cout << words << std::endl;
  }
  ```
- Hello from GCC 4.9.0 !
  ```

LAMBDA CALCUL (C++11)

- [capture](parameters) -> return_type { function_body }
- Closure between [ ]
  [] //no variables defined. Attempting to use any external variables in the lambda is an error.
  [x, &y] //x is captured by value, y is captured by reference
  [&] //any external variable is implicitly captured by reference if used
  [=] //any external variable is implicitly captured by value if used
  [&, x] //x is explicitly captured by value. Other variables will be captured by reference
  [=, &z] //z is explicitly captured by reference. Other variables will be captured by value
- ```
  std::vector<int> some_list{ 1, 2, 3, 4, 5 };
  int total = 0;
  std::for_each(begin(some_list), end(some_list), [&total](int x) {
        total += x;
  });
  ```
- ```
  std::vector<int> some_list{ 1, 2, 3, 4, 5 };
  int total = 0; int value = 5;
  std::for_each(begin(some_list), end(some_list), [&, value, this](int x) {
        total += x * value * this->some_func();
  });
  ```
- http://stackoverflow.com/questions/7627098/what-is-a-lambda-expression-in-c11

Java

CODE ORGANIZATION

- Each Java class is in a single file (except if nested)
- ".java" files
- Bytecode
  - packaged in "jar" files
  - executed with CLASSPATH

LiNACS
Embedded Linux project accelerator

LIBRARIES

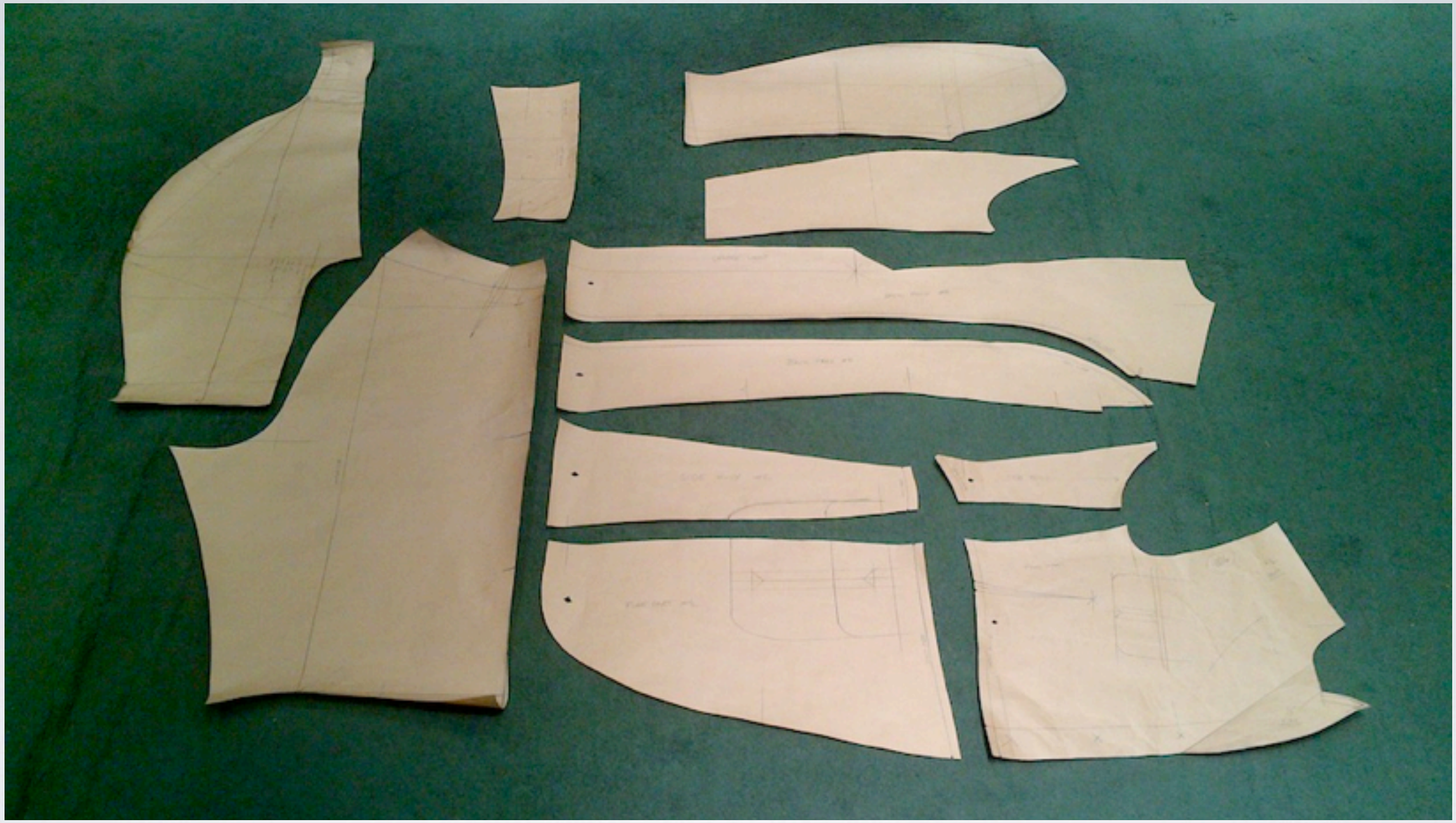
- Standard library with language
- Lot of different interpreters (only one official)
- Packages/sub-packages to import (as modules)
- Lot of tools
- Rather bad documentation

LiNACS
Embedded Linux project accelerator

STRENGTHS AND WEAKNESSES

- Rather easy to code, less mindfuck than C++
- Library not very difficult to handle (less possibilities, too)
- Easy to create threads (but very badly documented)
  - a thread is a new object Thread declare on-the-fly, nested (just override "run()" method and call "start()" on the create object after affectation)
  - keyword " synchronized" to protect objects and methods by mutex
- Nested classes and functions are cool
- Tends to be rather quickly messy and impossible to maintain

LiNACS
Embedded Linux project accelerator



DESIGN PATTERNS

WHAT IS IT (FOR)?

- General reusable solution to a commonly occurring problem within a given context in software design
- Ideas of code structure, best practices, architecture thinking
- Looooooot of patterns: http://en.wikipedia.org/wiki/Software_design_pattern
- Actually, a lot of theory, few applicable...
- hereafter, most useful ones: factory, functor, singleton and visitor

FACTORY

- Create your objects on demand
- Creational pattern: creating objects without specifying the exact class of object that will be created (not calling the constructor)
- Method: define a factory class (and create a factory object) with a method that return objects of different classes (which are children of the same base class), depending on arguments (e.g. a string or an enum)

LiNACS
Embedded Linux project accelerator

FUNCTOR

- object-function
- an object you call with ( ) operator (like a function)
- kind of (poor) lambda calculus
- ```
  List<String> list = Arrays.asList("10", "1", "20", "11", "21", "12");
  Comparator<String> numStringComparator = new Comparator<String>() {
      public int compare(String str1, String str2) {
          return Integer.valueOf(str1).compareTo(Integer.valueOf(str2));
      }
  };
  Collections.sort(list, numStringComparator);
  ```
- ```
  struct IntComparator {
    bool operator()(const int &a, const int &b) const  {  return a < b;  }
  };
  template <class RandomIt, class Compare> void sort(RandomIt first, RandomIt last, Compare comp);
  int main()
  {
      std::vector<int> items { 4, 3, 1, 2 };
      std::sort(items.begin(), items.end(), IntComparator());
      return 0;
  }
  ```

SINGLETON

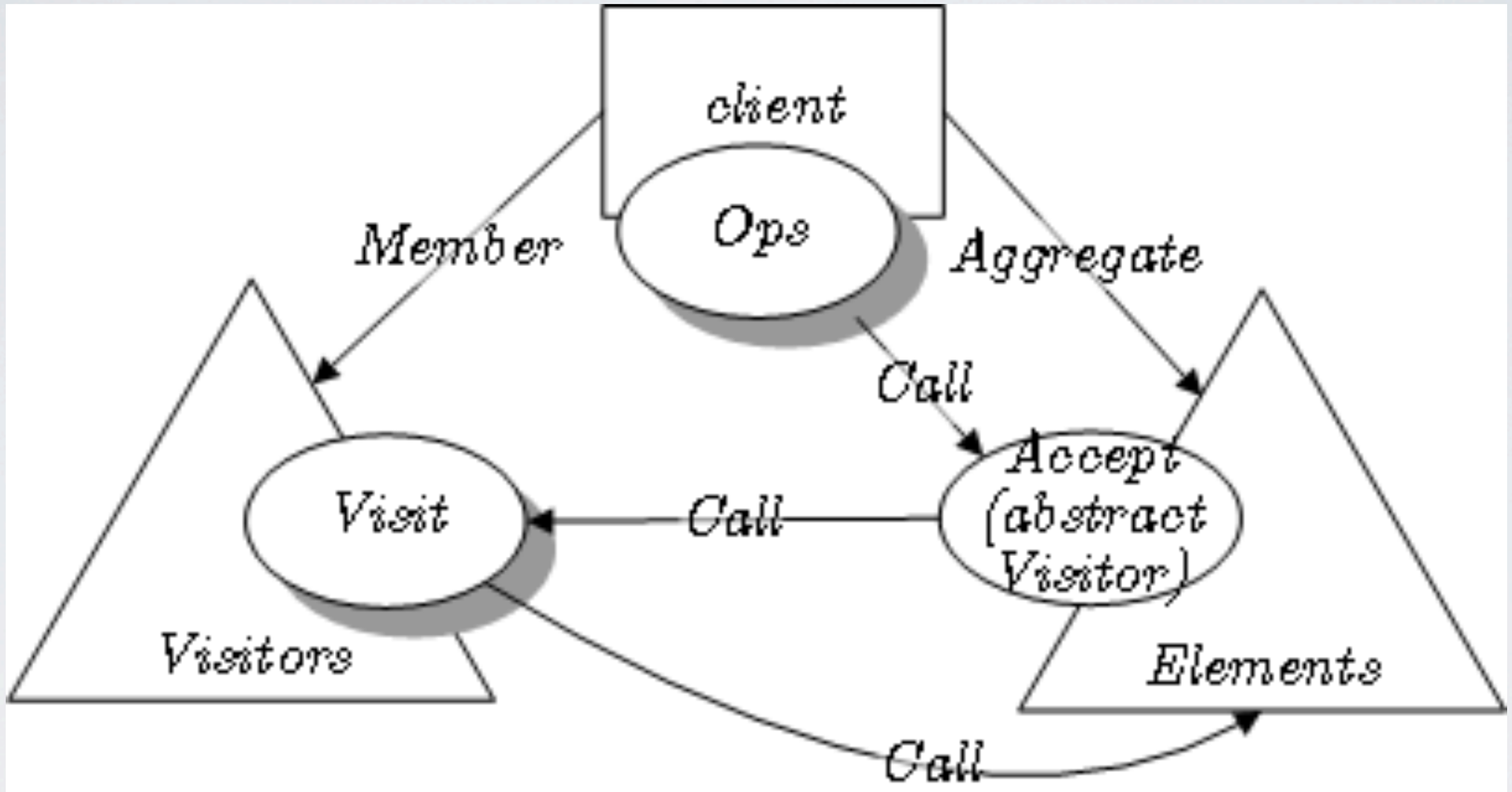
- Single instance of an object
- The singleton pattern is a design pattern that restricts the instantiation of a class to one object
- Very useful but... quite impossible to implement correctly in Java and C++!
- Can be an anti-pattern if used to much (similar to using to many globals...)

LinACS
Embedded Linux project accelerator

VISITOR

- Way of separating an algorithm from an object structure on which it operates
  - visitor object visits several objects accepting the visitor as input
  - the visitor takes the instance reference as input, and implements the goal through double dispatch
- Useful for making the same iterative action with different behavior for each object of different classes (e.g. linearizing in XML, Json, etc.)
- allows one to add new virtual functions to a family of classes without modifying the classes themselves

LiNACS
Embedded Linux project accelerator


client
Ops
Member
Aggregate
Call
Accept
(abstract
Visitor)
Visit
Call
Visitors
Elements
Call

```
#include <string>
#include <list>
#include <iostream>
#include <algorithm>
using namespace std;

class Pewpew;
class Visitor {
public:
    virtual void visit(Pewpew *p) = 0;
};

class Pewpew {
private:
    string name;
    int age, size, weight;
public:
    Pewpew(string name, int age, int size, int weight) :
         name(name), age(age), size(size), weight(weight) {}
    string getName() { return name; }
    int getAge() { return age; }
    int getSize() { return size; }
    int getWeight() { return weight; }
    void accept(Visitor *v) { v->visit(this); }
};
```

```
class CSVvisitor: public Visitor
{
public:
    void visit(Pewpew *p) {
        cout << p->getName() << ";" << p->getAge()
             << ";" << p->getSize() << ";" << p->getWeight() << endl;
    }
};

class Classpewpews
{
private:
    list<Pewpew> pewpews;
public:
    Classpewpews() {
        pewpews.push_back(Pewpew("Kevin", 21, 175, 70));
        pewpews.push_back(Pewpew("Jennifer", 20, 164, 50));
    }
    void saveCSV() {
        CSVvisitor v;
        for_each(pewpews.begin(), pewpews.end(),
                 [&](Pewpew &p) { p.accept(&v); });
    }
};

int main()
{
    Classpewpews c;
    c.saveCSV();
    return 0;
}
```

ITERATOR

- An ad-hoc object created to iterate on values stored in another container object
- This special iterator object takes all the different values from the attached container object as it is iterated
- The iterator can have special methods and properties allowing to access advance features (instead of using a function to decode the different values contained in the iterated object)
- Widely used in STL, but you can create your powerful ones

LiNACS
Embedded Linux project accelerator

ANTI-PATTERN

(THX WIKIPEDIA)

- Anemic Domain Model: The use of the domain model without any business logic. The domain model's objects cannot guarantee their correctness at any moment, because their validation and mutation logic is placed somewhere outside (most likely in multiple places).
- BaseBean: Inheriting functionality from a utility class rather than delegating to it
- Call super: Requiring subclasses to call a superclass's overridden method
- Circle-ellipse problem: Subtyping variable-types on the basis of value-subtypes
- Circular dependency: Introducing unnecessary direct or indirect mutual dependencies between objects or software modules
- Constant interface: Using interfaces to define constants
- God object: Concentrating too many functions in a single part of the design (class)
- Object cesspool: Reusing objects whose state does not conform to the (possibly implicit) contract for re-use
- Object orgy: Failing to properly encapsulate objects permitting unrestricted access to their internals
- Poltergeists: Objects whose sole purpose is to pass information to another object
- Sequential coupling: A class that requires its methods to be called in a particular order
- Yo-yo problem: A structure (e.g., of inheritance) that is hard to understand due to excessive fragmentation

LiNACS
Embedded Linux project accelerator

CODING TIME

code your own (very little) C++ library!

FIRST STEP: LIST

- { container ; next -> { container ; next -> { ..... -> NULL } .... } }
- Code it in C... just kidding (horrible)
- create a C++ class "List"
- List of int
- pointer to same class (private data)
- constructor & destructor
- method "append()" and operator<<
- method "remove()"
- methods "pop()", "pop_back()", "back()", "front()"
- method "at()" and "operator[]"
- make it generic: List of anything (template!)

LiNACS
Embedded Linux project accelerator

SECOND STEP: VECTOR

- [container, container, container, ...]
- create C++ class Vector
- difference with List is that size is already known from the beginning
- use C-style array (private data)
- constructor & destructor
- code same method than List
- may realloc() be with you!

LiNACS
Embedded Linux project accelerator

THIRD STEP: CONTAINER

- abstract C++ class
- List and Vector inherit Container
- Let's do some refactoring!
  - declare as virtual all same named methods
  - put methods code in common when possible, by factorization (example: "pop_back()" calling "remove()")

LiNACS
Embedded Linux project accelerator

PROJECT

Zelda coding

THE GAME

- You are princess Zelda and this time, it's your turn to save Links!
- The map is a labyrinth as in all other Zeldas:
  - there are some sprites you cannot pass (stones, walls, deep water...)
  - there are some sprites that can slow you (sand, water)
  - there are some sprites that allow you to go elsewhere (house, cave...)
  - hint: make a map editor!
- You can grab items, coins, life points, weapons, etc
- There are monsters that can heart you
  - they attack you, but you can also attack them!
  - you can die (sad)
  - you can gain new life points (hearts)
- There are some personages you can interact with (merchant, helpful people...)
- You can save and load the game

LiNACS
Embedded Linux project accelerator

HOW TO CODE

- In C++, with Qt5
  - using OOP precepts
  - using GUI precepts
- Start a github account (be careful, it will be public!)

EVALUATION

- Two notes, different scopes:
  - fun part:
    - how much do you have coded?
    - is the game entertaining?
    - how many bugs?
  - technical part:
    - OOP: first note
    - GUI: second note
- IRL evaluation

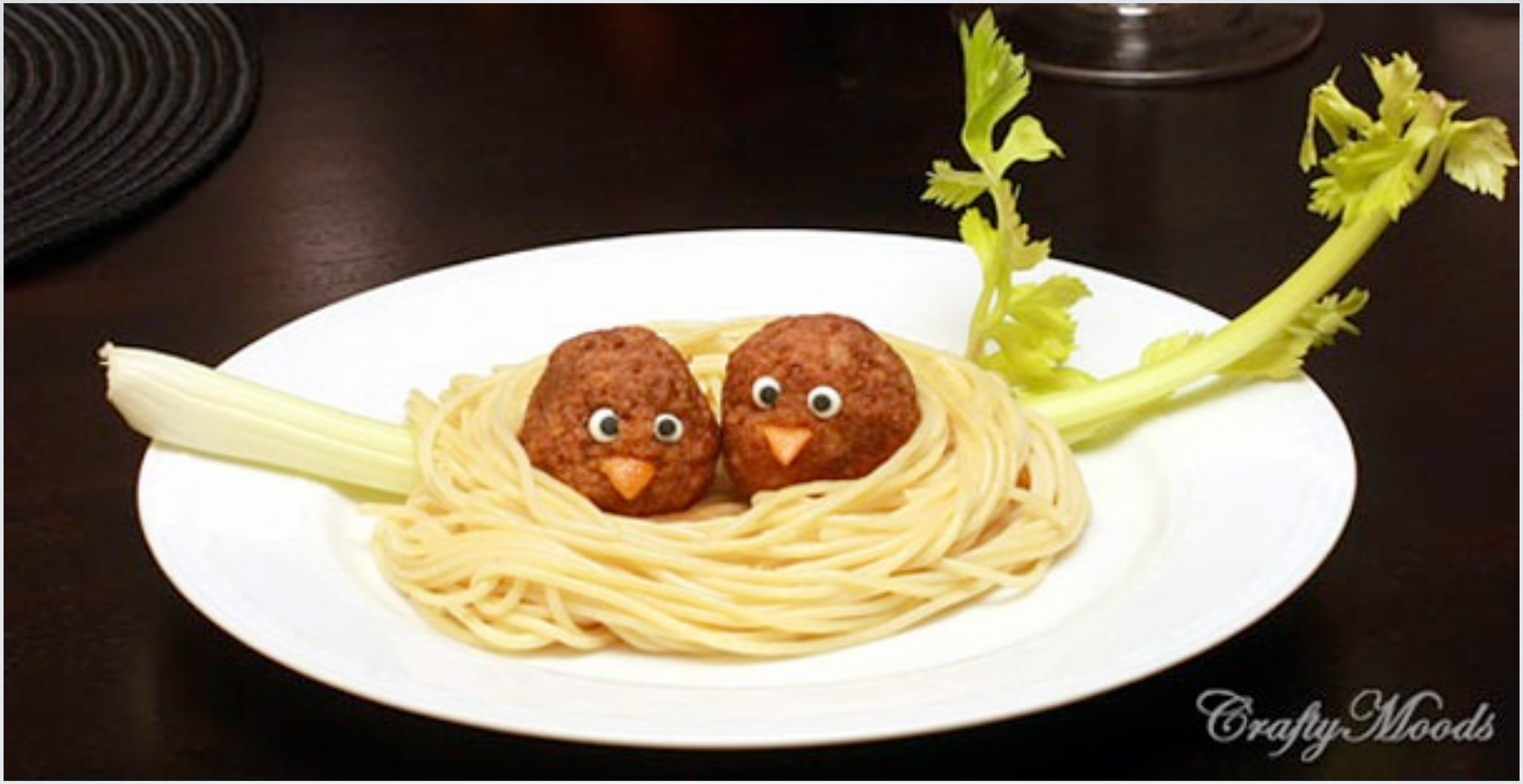
LiNACS
Embedded Linux project accelerator

SCHEDULE

- BEGIN NOW!!!!
- On October 19th midnight, make a tag, send me a tarball
  - if it does not compile: 0/20
  - if it crashes at launch: 0/20
- Special extra time of one week: October 26th midnight
  - special price: you lose 4 points on you note

AND REMEMBER...

HAVE FUN!!!!!


CraftyMoods

THANKS!

Gilles BLANC — LINACS
http://linacs-consulting.com
gilles.blanc@linacs-consulting.com
+33 (0)6 70 33 00 03