**Elegant Objects**

Def : **Elegant Objects** (EO) is an object-oriented programming paradigm that renounces traditional techniques like [null](http://www.yegor256.com/2014/05/13/why-null-is-bad.html), [getters-and-setters](http://www.yegor256.com/2014/09/16/getters-and-setters-are-evil.html), [code in constructors](http://www.yegor256.com/2015/05/07/ctors-must-be-code-free.html), [mutable objects](http://www.yegor256.com/2014/06/09/objects-should-be-immutable.html), [static methods](http://www.yegor256.com/2014/05/05/oop-alternative-to-utility-classes.html), [annotations](http://www.yegor256.com/2016/04/12/java-annotations-are-evil.html), [type casting](http://www.yegor256.com/2015/04/02/class-casting-is-anti-pattern.html), [implementation inheritance](http://www.yegor256.com/2016/09/13/inheritance-is-procedural.html), [data objects](http://www.yegor256.com/2016/07/06/data-transfer-object.html), [etc.](http://www.yegor256.com/2014/09/10/anti-patterns-in-oop.html)

**Birth :**

Never use -er names :

such names don’t convey what these objects are, but rather what they do

Une image contenant texte

Description générée automatiquement

### Make one constructor primary :

### An additional benefit of the second one is to have clear names defining, for example, states that an object can have

### Une image contenant texte Description générée automatiquement

### Keep constructors code-free :

### Having such an abstraction makes it easier to embed functionality later to this class.

### Une image contenant texte Description générée automatiquement

### Education :

### Encapsulate as little as possible :

### the author encourages using as few attributes encapsulated per class as possible, arguing that if your class has many, it probably means that some of those properties could be composed into smaller parts, building therefore a tree of objects.

### Encapsulate something at the very least :

### if you want to represent something, this thing should have a state to represent it

### Always use interfaces :

### interfaces define behavior (public methods), so if a class has a behavior, it should implement an interface that defines it

### Choose method names carefully :

### Builders: methods that create or build something.

### Manipulators: methods that change something in the real world.

### Don’t use public constants :

### if you have public constants, you will introduce coupling, because other classes will use these public constants, as well as lose cohesion, since they now depend on another source.

### Be immutable :

### mutability vs. Immutability. Immutable objects should be used whenever possible, and whenever it does not imply a complete rewrite of big chunks of your system

### Write tests not documentation :

### if your code is simple, and well tested, and your code and tests are clear, ou have instructions and a proof of what your code does.

### Don’t mock; use fakes :

### And that this fake should live inside the interface itself, something like this:

### Une image contenant texte Description générée automatiquement

### Keep interfaces short; use smarts :

### a Smart is defined as an inner class of an interface, where this inner class defines methods that extends the capabilities of the interface.

## **Retirement :**

### Never return NULL

### Throw only checked exceptions

### Be either final or abstract

### Use RAII