# Elegant Objects: Developing a Programming Language for Large-Scale Applications


## Abstract

Object-oriented programming (OOP) is one of the most popular programming paradigms used for building software systems. However, despite its industrial and academic popularity, OOP is widely criticized for its high complexity, low maintainability, and lack of agreed-upon and rigorous principles. Eolang (a.k.a. EO) was created to solve the said problems by restricting language features and introducing a formal object calculus and a programming language with a reduced set of features.


## Introduction

The fundamental problem in OOP is the lack of a rigorous formal model, the high complexity, the too many ad hoc designs, and the fact that programmers are not satisfied. Many OOP languages,including Java, have critical design flaws. Due to these fundamental issues, many Java-based software products are low quality and hard to maintain.
The above drawbacks often result in system failures, customer complaints, and lost profits. The problem has been recognized; however, it has not been addressed yet.

Elegant Objects, is the vision of pure that is free from the incorrectly taken design decisions common for the mainstream technologies. Specifically, these are: Static methods and attributes; Classes; Implementation inheritance; Mutable objects; Null references; Global variables and methods; Reflection and annotations; Typecasting; Scalar data types; Flow control operators (for loop, while loop, etc.).

## Research Problem and Objectives

This work aims to analyze the features of EOLANG semantics based on the available information. The main task of this analysis is to assess the reliability of the proposed language solutions. We also analyze approaches to building a code generator that provides an acceptable efficiency of transformation of language constructions. In addition, an attempt is made to propose alternative solutions at the level of the programming language, which will improve the reliability of the development process, as well as a more efficient implementation of the compiler.

# Discussion

This section attempts to analyze the features of EOLANG semantics based on the available information. The main task of this analysis is to assess the reliability of the proposed language solutions. We also analyze approaches to building a code generator that provides an acceptable efficiency of transformation of language constructions. In addition, an attempt is made to propose alternative solutions at the level of the programming language, which will improve the reliability of the development process, as well as a more efficient implementation of the compiler.

## EO Abstract objects

*An object is **abstract** if at least one of its attributes is free—isn't bound to any object. An object is **closed** otherwise.*

*--Yegor Bugaenko*

The term "Abstract Object" itself contains a semantic contradiction. On one hand, the object seems to exist, but on the other hand, there is uncertainty in it, which in many cases does not allow working with it normally. That is, using such an object, you can try to access the data that has not yet been initialized. This leads to the interruption of the program during execution after a successful compilation. Many of these errors are difficult to detect, because when accessing an object using methods, it is not obvious whether it contains undefined attributes or they are absent in this method. The situation may be aggravated by the fact that in the language it is possible to produce new abstract objects with partial definition from other abstract objects.

### Example 1. Demonstration that abstract objects reduce the reliability of programming

File with rectangle

```
+package rectangle2
+alias stdout org.eolang.io.stdout
+alias sprintf org.eolang.txt.sprintf

[a b] > rectangle

  sprintf > toString
    "It is Rectangle: a = %d, b = %d"
    a
    b

  mul. > square
    a
    b
```

```
mul. > perimeter
  add.
    a
    b
  2

stdout > out
  sprintf
    "It is Rectangle: a = %d, b = %d\n"
    a
    b

[r] > cmpSquare
  if. > @
    eq.
      r.square
      ^.square
    "Yes"
    "No"
```

File with options for using a rectangle

```
+package rectangle2
+alias rectangle rectangle2.rectangle
+alias stdout org.eolang.io.stdout
+alias sprintf org.eolang.txt.sprintf

[args...] > app

  rectangle > r
    (args.get 0).toInt
    (args.get 1).toInt

  rectangle > r2
    3
    4

  rectangle > r3
    13:b
    42:a

  rectangle > r4
    13:b

  rectangle > r5
    1313

  rectangle > r6

  r3 > r7

  r7 > r8
```

```
  555

r8 > r9
  101010:b

seq > @

  r.out

  r.square > s!
  stdout
    sprintf
      "Square = %d\n"
      s

  r.perimeter > p!
  stdout
    sprintf
      "Perimeter = %d\n"
      r.perimeter

  stdout
    sprintf
      "r2: a = %d, b = %d\n"
      r2.a
      r2.b
  r2.out

  stdout
    sprintf
      "r3: a = %d, b = %d\n"
      r3.a
      r3.b
  r3.out

  stdout
    sprintf
      "r4: b = %d\n"
      r4.b

  stdout
    sprintf
      "r5: a = %d\n"
      r5.a

  stdout
    sprintf
      "r6: a = ???\n"

  stdout
    sprintf
      "r7 as clone r3: a = %d, b = %d\n"
      r7.a
      r7.b
  r7.out
```

```
stdout
  sprintf
    "r8 as clone r7: a = %d, b = %d\n"
    r8.a
    r8.b
r8.out

stdout
  sprintf
    "r9 as clone r8: a = %d, b = %d\n"
    r9.a
    r9.b
r9.out
```

Results of work

```
It is Rectangle: a = 3, b = 5
Square = 15
Perimeter = 16
r2: a = 3, b = 4
It is Rectangle: a = 3, b = 4
r3: a = 42, b = 13
It is Rectangle: a = 42, b = 13
r4: b = 13
r5: a = 1313
r6: a = ???
r7 as clone r3: a = 42, b = 13
It is Rectangle: a = 42, b = 13
r8 as clone r7: a = 555, b = 13
It is Rectangle: a = 555, b = 13
r9 as clone r8: a = 555, b = 101010
It is Rectangle: a = 555, b = 101010
```

The example demonstrates that the use of partially defined (abstract) objects violates the integrity of their perception and functioning. Using object methods as well as undefined attributes leads to runtime errors. Only partial direct access to initialize attributes is possible. Almost all methods directly related to the processing of the state of the object cease to work.

## EO Type system analysis

One of the features of the language is the lack of typing. That is, if we consider the original description, then the type system (both static and dynamic) is absent. Achieving the uniqueness of language constructions is determined mainly at the level of operational unambiguity, when the semantics of the operations performed is revealed as a combination of data objects and method objects. That is, there is a sense of analogy with assembly languages, in which the location of data in memory

is not tied to the type, and the semantics of command execution is determined mainly by the operation code and, in some cases, by additional data identification through registers and memory access.

*It should be noted that additional identification appeared in later assemblers, such as x86 assembler. Before that, many assemblers got by with identification through an opcode (for example, in pdp-11)*

As a result, the use of implicit type conversions leads to a programming style that is often reminiscent of assembly. Before performing one or another operation defined by an object, you have to explicitly convert data to the desired object (equivalent to type conversion) or determine the semantics of operations using a constant (constant object) used as an argument.

Therefore, the type of object being processed is often impossible to predict. This is especially true when passing parameters. It is easy to generate errors related to the fact that instead of an object of one type, an object of a different type is passed for processing. The transformations of the transmitted object can be hidden by their location in other files or in some other way. You have to use an explicit cast at the point of object processing. However, there are problems associated with the fact that not all types of objects can be explicitly transformed. That is, in fact, we have a typeless representation of objects when passing parameters.

Example 2. Demonstration of what the lack of typing leads to

```
+package book2
+alias stdout org.eolang.io.stdout
+alias sprintf org.eolang.txt.sprintf

[args...] > app
  seq > @
    stdout "Hello, World!\n"
    transform 5

    [isbn] > book2
      "Object Thinking" > title
      memory > price
      seq > @
        stdout
          sprintf "title: %s\n"
            title
        price.write 10.0
        stdout
          sprintf "price = %f\n"
            price
        price.write 200
        stdout
```

```
      sprintf "price = %d\n"
        price
    price.write "Hello, memory!"
    stdout
      sprintf "price = %s\n"
        price

  book2 "cool book" > bbb
  stdout
    sprintf "bbb isbn = %s\n"
      bbb.isbn
  stdout
    sprintf "title = %s\n"
      bbb.title
  book2.price.write 1000
  stdout
    sprintf "book 2 price = %d\n"
      book2.price
  stdout
    sprintf "bbb price = %s\n"
      bbb.price

  book2 666 > xxx
  stdout
    sprintf "xxx isbn = %d\n"
      xxx.isbn

  book2 2.71828 > yyy
  stdout
    sprintf "yyy isbn = %f\n"
      yyy.isbn
[a] > transform
  stdout > @
    sprintf "%d\n" a
```

Results of work

```
  Hello, World!
  5
  title: Object Thinking
  price = 10,000000
  price = 200
  price = Hello, memory!
  title: Object Thinking
  price = 10,000000
  price = 200
  price = Hello, memory!
  bbb isbn = cool book
  title = Object Thinking
  book 2 price = 1000
  bbb price = Hello, memory!
  title: Object Thinking
  price = 10,000000
  price = 200
```

```
price = Hello, memory!
xxx isbn = 666
title: Object Thinking
price = 10,000000
price = 200
price = Hello, memory!
yyy isbn = 2,718280
```

The example shows that a typeless solution leads to the dualism of using the same attributes. This leads to the fact that even generated objects become incomparable with each other. Especially when the connection between them is not tracked in the future. It should also be noted that cloning leads to copying the internal states of objects, which is not always advisable.

Identity (originating from one) of objects can be modeled using a common constant that identifies the type of the object. But this is again a purely assembler trick that does not allow object matching at compile time.

## EO Processing alternatives

The handling of alternative objects that have similar behavior is one of the common situations in dynamic linking. In the procedural approach, alternatives are usually analyzed explicitly either by checking the attached type (explicit indication of the type of the alternative is used, for example, in the union of the Ada programming language), or by using a sign (key) explicitly entered by the programmer (for example, in the C language), which simulates the variant notation programming language Pascal.

In the case of the OO approach, explicit type checking in most cases is replaced by "duck typing" in which any object that executes some predefined method, also existing in other objects, can execute its code, which is determined by the internal implementation of this method. In this case, OO has polymorphism, which in most modern OO languages is implemented through the inheritance (or extension) mechanism, when the methods of the base class are overridden in derived classes. This approach is usually implemented through virtual method tables, which provide reasonably fast access to virtual methods.

Another option for implementing polymorphism is to use associative access (associative dynamic polymorphism), when an object containing some methods, having received a message from another object, looks for a method in its method table. Having matched the received message by a model and having found the appropriate method, it carries out its execution. If absent, the exception is handled. This approach is much slower than the previous one. However, it allows, through the identity of the methods, to bind and launch objects that are not otherwise related to each other.

It can also be noted that support for polymorphism is currently possible in procedural languages as well. However, it has been implemented using different principles. In particular, Go duck typing is implemented through the introduction of interfaces. A similar mechanism is used in the Rust programming language. Also, in a number of works (mine), procedural-parametric polymorphism was proposed and experimentally demonstrated on the extension of the Oberon-2 language.

It is also worth noting that all types of polymorphism, in principle, can be implemented in functional programming languages.

EOLANG uses associative dynamic polymorphism, where unrelated objects can execute their methods with the same signature, connecting to some object. This connection can be mutable or random. Then it is not known which of the objects is connected at the moment. And only by its visible behavior (and it may not always be visible) it is possible to determine which of the objects is currently connected.

### File with polymorphism

```
+package fig01
+alias rectangle fig01.rectangle
+alias triangle fig01.triangle
+alias stdout org.eolang.io.stdout
+alias sprintf org.eolang.txt.sprintf

[f] > figure
  stdout > out
    sprintf "Figure. %s\n"
      f.toString

[args...] > app

  rectangle 10 20 > r
  triangle 3 4 5 > t
  figure r > figR
  figure t > figT
  memory > m

  seq > @

    r.out
    r.perimeter > pr!
    stdout
      sprintf
        "Perimeter = %d\n"
        pr

    t.out
    t.perimeter > pt!
    stdout
      sprintf
        "Perimeter = %d\n"
        pt
```

```
  figR.out
  figT.out
```

File with a triangle

```
+package fig01
+alias stdout org.eolang.io.stdout
+alias sprintf org.eolang.txt.sprintf

[a b c] > triangle
  2 > @
  add. > perimeter
    add.
      a
      b
    c

  stdout > out
    sprintf
      "It is Triangle: a = %d, b = %d, c = %d\n"
      a
      b
      c

  sprintf > toString
    "It is Triangle: a = %d, b = %d, c = %d"
    a
    b
    c

  [t] > cmpSquare
    if. > @
      eq.
        t.square
        ^.square
      "Yes"
      "No"
```

File with a rectangle

```
+package fig01
+alias stdout org.eolang.io.stdout
+alias sprintf org.eolang.txt.sprintf

[a b] > rectangle
  1 > @
  sprintf > toString
    "It is Rectangle: a = %d, b = %d"
    a
```

```
  b

mul. > perimeter
  add.
    a
    b
  2

stdout > out
  sprintf
    "It is Rectangle: a = %d, b = %d\n"
    a
    b

[r] > cmpSquare
  if. > @
    eq.
      r.square
      ^.square
    "Yes"
    "No"
```

## Results of work

The presence of duck typing cannot always be used to accurately identify an object. This is due to the fact that in some cases a preliminary type analysis is required before applying the methods. The lack of explicit type checking in the language makes it impossible to identify the connected object. However, this check can be implemented programmatically by mapping to each of the objects an appropriate constant or method that returns the type of the object. Typically, even statically typed languages use dynamic run-time type checking for objects derived from derived classes.

In a dynamically typed system, the type of an object is also implicitly specified. However, this type is stored inside an object and can be explicitly checked at runtime.

In order to explicitly check the type of an object, it is necessary to introduce an additional attribute into it, which is returned by the specified attribute. For example, an integer. Then the comparison of objects for similarity is carried out using the same value of this attribute.

Duck typing used in the language is based on the similarity of methods for processing objects, regardless of their relationship, for example, through the mechanism of inheritance. This allows you to connect a variety of objects, which leads to a decrease in the reliability of the program code. In addition, for such a connection, a mechanism based on associative arrays is usually used, which slows down the calling of methods (the call is carried out by name with the search for the

desired name in the map).

It should also be noted that the typeless mechanism allows you to call methods with the same names, but used for completely different purposes, which also leads to a decrease in the reliability of the program.

Unfortunately, it was not possible to implement the example with memory. When we wanted to use polymorphism with the same mutable object, connecting sequentially different shapes to it and calling a polymorphic method. Apparently, in the current version, polymorphic work through memory is not implemented. Although in many programs there is just a substitution of one object for another.

# Possible approaches

## Abstract objects

From one point of view it is necessary to prohibit the use of abstract objects in the executable code, since their use reduces the reliability of the program. However, by themselves, abstractions that are undefined in meaning can be useful. Therefore, one can simply return to the concept of a class. In principle, it seems to be not so bad with a few variants of class constructors that provide partial initialization, although here, too, partial initialization can lead to a decrease in programming reliability. Also, nothing prevents keeping the cloning of objects at the level of language support, thereby eliminating the need to implement the Prototype pattern.

## Type system analysis

Introduce a static type system into the programming language, similar to modern OO programming languages, with possible enhancements to improve data control at compile time. As an addition at subsequent stages of the language development for methods with static polymorphism (method name overloading), you can introduce type parsing and type inference, characteristic of pure functional programming languages. Perhaps in some kind of truncated version.

The introduction of static typing will improve the reliability of programs and provide additional control both during compilation and when using static code analyzers.

## Processing alternatives

The use of associative dynamic polymorphism (ADP) in the language allows flexible substitution of objects, using the call for different objects of the same message. However, it should be noted that this method is more suitable for dynamically typed languages. The use of a typeless organization of objects in the language (when it is

impossible to explicitly and specifically indicate or determine the type) leads to a decrease in the reliability of the software being developed.

As a variant of the development of a language that uses the implemented approach to duck typing, we can propose the addition of dynamic typing of objects. Then, if necessary, you can explicitly check the type of the object without explicitly using the corresponding constants. However, in this case too, excessive flexibility in the organization of polymorphism can lead to a decrease in the reliability of the code and its control due to the substitution of any object whose method signature allows you to do this. On the other hand, ADP is slow enough to be used in real-time systems.

More reliable, in my opinion, is the use of static typing in combination with methods for supporting polymorphism oriented towards this, which allows to control the range of connected objects. It is also advisable to use a dynamic type checking mechanism at runtime, which is usually implemented in almost all OO languages.

## Conclusion

The previously described semantic model of the language and its equivalent presentation in procedural form demonstrates the features EOLANG on the organization of the development process. However, they also show the limited language, which leads to a decreased reliability of the developed code. Lack of some instrumental support requires writing additional code not directly related to the problem being solved (with the subject area and its algorithms).
In this connection, it might be suitable to analyze the possibilities extensions of the semantics of the language focused on additional instrumental support for methods, increasing the efficiency and reliability of development software.

## References

...