Reducing Programs to Objects

Ver: `0.0.0`

Yegor Bugayenko
Huawei
Russia, Moscow
yegor256@gmail.com

Abstract

C++, Java, C#, Python, Ruby, JavaScript are the most powerful object-oriented programming languages, if language power would be defined as the number of *features* available for a programmer. EO, on the other hand, is an object-oriented programming language with a reduced set of features: it has nothing by objects and mechanisms of their composition and decoration. We are trying to answer the following research question: "Which known features are possible to implement using only objects?"

1 Features

To answer our research question we selected most complex features and demonstrated how each of them may be represented in EO (Bugayenko, 2021) objects[1]:

- Non-conditional jumps (Sec. 1.1),
- Data and code pointers (Sec. 1.2),
- Procedures (Sec. 1.3),
- Classes (Sec. 1.4),
- Exceptions (Sec. 1.6),
- Anonymous functions (Sec. 1.7),
- Generators (Sec. 1.8),
- Types and casting (Sec. 1.9),
- Reflection (Sec. 1.10),
- Static methods (Sec. 1.11),
- Inheritance (Sec. 1.12),
- Method overloading (Sec. 1.13),
- Java generics (Sec. 1.14),
- C++ templates (Sec. 1.15),
- Mixins (Sec. 1.16),
- Java annotations (Sec. 1.17).

Other features are more trivial, that's why they are not presented in this paper, such as operators, loops, variables, code blocks, constants, branching, and so on.

[1] LaTeX sources of this paper are maintained in REPOSITORY GitHub repository, the rendered version is `0.0.0`.

1.1 Goto

Goto is a one-way imperative transfer of control to another line of code. There are only two possible cases of goto jumps: forward and backward.

1.1.1 Backward Jump. This is a "backward" jump example in C language:

```
#include "stdio.h"
void f() {
  int i = 1;
  again:
  i++;
  if (i < 10) goto again;
  printf("Finished!");
}
```

It can be mapped to the following EO code:

```
[] > f
  memory 0 > i
  seq > @
    i.write 1
    goto
      [g]
        seq > @
          i.write (i.plus 1)
          if.
            i.lt 10
            g.backward
            TRUE
    QQ.io.stdout "Finished!"
```

Here, the one-argument abstract atom `goto` is being copied with a one-argument abstract anonymous object, which is the sequence of objects doing the increment of `i` and then the comparison of it with the number ten. If the condition is true, `g.backward` is called, which leads to a backward jump and re-iteration of `goto`.

1.1.2 Forward Jump. This is an example of a "forward" jump in C language:

```
int f(int x) {
  int r = 0;
  if (x == 0) goto exit;
  r = 42 / x;
  exit:
  return r;
}
```

It can be mapped to the following EO code:

```
[x] > f
  memory 0 > r
  seq > @
    r.write 0
    goto
      [g]
        seq > @
          if.
            x.eq 0
            g.forward TRUE
            TRUE
          r.write (42.div x)
    r
```

Here, the same abstract atom `goto` is copied with an abstract one-argument object, which is a sequence of objects. When the condition is true, a forward jump is performed by `g.forward` atom.

Similar way, the atom `goto` may be used to simulate other conditional jump statements, like `break`, `continue`, or `return` in the middle of a function body (see Sec. 1.3).

1.1.3 Complex Case. This is a more complex case of using `goto` in C:

```
#include "stdio.h"
void f(int a, int b) {
  goto start;
back:
  printf("A");
start:
  if (a > b) goto back;
  printf("B");
}
```

In order to translate this code to EO it has to be refactored as Fig. 1 demonstrates. The function `f()` is copied twice and each copy has its own execution flow implemented.

```
#include "stdio.h"
void f(int a, int b) { f1(a, b); }
void f1(int a, int b) {
```

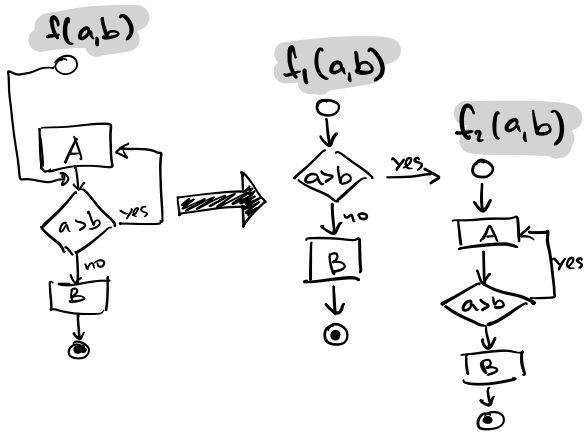

Figure 1. The function `f` with a few `goto` statements inside is translated to two functions, reducing the complexity of the code at the cost of introducing duplication.

```
  if (a > b) f2(a, b);
  printf("B");
}
void f2(int a, int b) {
back:
  printf("A");
  if (a > b) goto back;
  printf("B");
}
```

Then, the translation to EO is trivial, with the use of the `goto` object.

In more complex cases a program may first be restructured to replace `goto` statements with loops and branching, as suggested by Williams et al. (1985), Pan et al. (1996), Erosa et al. (1994), and Ceccato et al. (2008).

1.1.4 Multiple Returns. Some structured programming languages allow a function to exit at an arbitrary place, not only at the end, using `return` statements, for example:

```
void abs(int x) {
  if (x > 0) {
    return x;
  }
  return -1 * x;
}
```

This can be mapped to the following EO code using `goto` object:

```
[x] > abs
```

```
  goto > @
    [g]
      seq > @
        if.
          x.gt 0
          g.forward x
          TRUE
        g.forward
          -1.times x
```

The dataization of `g.forward` will exit the `goto` object wrapping the entire code in the "function" `abs`.

1.2 Pointers

A pointer is an object in many programming languages that stores a memory address. Pointers may point to data memory and to program memory.

1.2.1 Pointers to Data. This is an example of C code, where the pointer `p` is first incremented by seven times `sizeof(book)` (which is equal to 108) and then de-referenced to become a struct `book` mapped to memory. Then, the `title` part of the struct is filled with a string and the `price` part is returned as a result of the function `f`:

```
#include "string.h"
struct book {
  char title[100];
  long long price;
};
int f(struct book* p) {
  struct book b = *(p + 7);
  strcpy(b.title, "Object Thinking");
  return b.price;
}
```

This code can be mapped to the following EO code:

```
[ptr] > book
  ptr > @
  ptr.block > title
    100
    [b] (b.as-string > @)
  ptr.block > price
    8
    [b] (b.as-int > @)
pointer. > p1
  heap 1024
  0
  108
[p] > f
  seq > @
    book (p.add 7) > b
    b.title.write
      ("Object Thinking").as-bytes
    b.price
```

Here, `p1` is an object that can be used as an argument of `f`. It is a copy of an abstract object `heap.pointer`, which expects two: 1) an absolute address in memory, and 2) the size of the memory block it points to, in bytes. Its attribute `block` expects two attributes: 1) the number of bytes in the heap, and 2) an abstract object that can encapsulate `bytes` which were just read from memory.

The object `heap` is an abstraction of a random access memory.

1.2.2 Pointers to Code. A pointer may not only refer to data in the heap, but also to executable code in memory (there is no difference between "program" and "data" memories both in x86 and ARM architectures). This is an example of C program, which calls a function referenced by a function pointer, providing two integer arguments to it:

```
int foo(int x, int y) {
  return x + y;
}
int f() {
  int (*p)(int, int);
  p = &foo;
  return (*p) (7, 42);
}
```

This code can be mapped to the following EO code:

```
[x y] > foo
  x.plus y > @
[] > f
  cage 0 > p
  seq > @
    p.write
      [x y]
        foo x y > @
    p.@ 7 42
```

Important to notice that the following code is not possible to represent in EO:

```
int f() {
  int (*p)(int);
```

```
  p = (int(*)(int)) 0x761AFE65;
  return (*p) (42);
}
```

Here, the pointer refers to an arbitrary address in program memory, where some executable code is supposed to be located.

1.2.3 Variables in Stack. This C function, which returns seven (tested with GCC), assumes that variables are placed in stack sequentially and uses pointer de-referencing to access them:

```
int f() {
  long long a = 42;
  long long b = 7;
  return *(&a - 1);
}
```

This code can be mapped to the following EO code:

```
[p] > long64
  p.block > @
    8
    [b] (b.as-int > @)
[] > f
  seq > @
    malloc. > stack
      heap 32 > h
      16
    long64 stack > b
    b.write (7.as-bytes)
    long64 stack > a
    a.write (42.as-bytes)
    long64 (a.p.sub 1) > ret!
    h.free stack
    ret
```

Here, the atom `malloc` allocates a segment of bytes in the `heap`. Later, the atom `free` releases it. The attribute `ret` is made constant in order to avoid its recalculation after it's "returned" (this mechanism is explained in Sec. 1.5 where destructors are discussed).

1.3 Procedures

A subroutine is a sequence of program instructions that performs a specific task, packaged as a unit. In different programming languages, a subroutine may be called a routine, subprogram, function, method, or procedure. In this PHP example, a function `max` is defined:

```
function max($a, $b) {
  if ($a > $b) return $a;
  return $b;
}
```

It can be mapped to the following EO code:

```
[a b] > max
  goto > @
    [g]
      seq > @
        if.
          a.gt b
          g.forward a
          TRUE
        b
```

This example also demonstrates how `goto` object can be used to simulate the behavior of the `return` statement from within the body of a method.

1.3.1 Impure Functions. A function is pure when, among other qualities, it doesn't have side effects: no mutation of local static variables, non-local variables, mutable reference arguments or input/output streams. This PHP function is impure:

```
$a = 42;
function inc($a) {
  $a = $a + 1;
  return $a;
}
inc(inc($a)); // $a == 44
```

It may be mapped to a constant EO object:

```
memory 42 > a
[x] > inc!
  seq > @
    x.write
      x.plus 1
inc
  inc a
```

1.4 Classes

A class is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods). The following program contains a Ruby class with a single constructor, two methods, and one mutable member variable:

```
class Book
  def initialize(i)
    @id = id
    puts "New book!"
  end
  def path
    "/tmp/${@id}.txt"
  end
  def move(i)
    @id = i
  end
end
```

It can be mapped to the following EO code, where a class becomes a factory of objects:

```
[i] > book
  cage i > id
  [] > ruby-init
    QQ.txt.sprintf > @
      "New book!"
  [] > path
    QQ.txt.sprintf > @
      "/tmp/%s.txt"
      id
  [i] > move
    id.write i > @
```

Here, the constructor is represented by the object `ruby-init`, which initializes finishes the initization of the object. The making an "instance" of a book would look like this:

```
book 42 > b
b.ruby-init
b.move 7
b.path
```

1.5 Destructors

In C++ and some other languages, a destructor is a method that is called for a class object when that object passes out of scope or is explicitly deleted. The following code will print both `Alive` and `Dead` texts:

```
#include <iostream>
class Foo {
public:
  Foo() { std::cout << "Alive"; }
  ~Foo() { std::cout << "Dead"; };
};
int main() {
  Foo f = Foo();
}
```

It may be translated to EO as such:

```
[] > foo
  [] > constructor
    QQ.io.stdout "Alive" > @
  [] > destructor
    QQ.io.stdout "Dead" > @
[] > main
  foo > f
  seq > @
    f.constructor
    f.destructor
```

There is no garbage collection in EO, that's why a destructor must be explicitly "called" when an object passes out of scope or is deleted.

1.6 Exceptions

Exception handling is the process of responding to the occurrence of exceptions—anomalous or exceptional conditions requiring special processing—during the execution of a program provided. This C++ program utilizes exception handling to avoid segmentation fault due to null pointer de-referencing:

```
#include <iostream>
class Book { public: int price(); };
int price(Book* b) {
  if (b == NULL) throw "NPE!";
  return (*b).price() * 1.1;
}
void print(Book* b) {
  try {
    std::cout << "The price: " << price(b)
        ;
  } catch (char const* e) {
    std::cout << "Error: " << e;
  }
}
```

This mechanism may be implemented in EO:

```
[] > book
  [] > price /int
[b] > price
  if. > @
    b.eq 0
    error "NPE!"
    b.price.times 1.1
[b] > print
```

```
  try > @
    []
      QQ.io.stdout > @
        QQ.txt.sprintf
        "The price: %d"
        price b
    [e]
      QQ.io.stdout > @
        QQ.txt.sprintf
        "Error: %s"
        e
    []
      nop > @
```

Here, the object `try` expects three arguments: 1) an abstract object to be dataized, and 2) an abstract object to be copied with one argument, in case dataization returns encapsulated object, and 3) an object to be dataized anyway (similar to Java `finaly` block).

In the object `price` we get the object `error`, which if dataized, causes the termination of dataization and a non-conditional jump to the "catch" object of the `try`. The mechanism is similar to "checked" exceptions in Java, where a method's signature must declare all types of exceptions the method may throw.

1.6.1 Many Exception Types.

A Java method may throw a number of either checked or unchecked exceptions, for example:

```
void f(int x) throws IOException {
  if (x == 0) {
    throw new IOException();
  }
  throw new RuntimeException();
}
```

This would be represented in EO as such:

```
[x] > f
  if. > @
    x.eq 0
    error "IOException"
    error "RuntimeException"
```

To catch both exceptions the object `f` would be used like this:

```
try
  []
    try > @
      []
        f 5 > @
      [e1]
        QQ.io.stdout e1 > @
      []
        nop > @
  [e2]
    QQ.io.stdout e2 > @
  []
    nop > @
```

1.7 Anonymous Functions

Anonymous functions that can be passed into methods as arguments. For example, in this Ruby code a "block" (a name Ruby uses for anonymous functions) is passed:

```
def scan(lines)
  lines.each do |t|
    if t.starts_with? '#' yield t
  end
end
scan(array) { |x| puts x }
```

This mechanism may be implemented in EO:

```
[lines b] > scan
  lines.each > @
    [t]
      if. > @
        t.starts-with "#"
        b t
        TRUE
scan
  array
  [x]
    QQ.io.stdout x > @
```

Here, the anonymous function passed to the object `scan` as an argument `b`. The "call" of this function is the dataization of its copy, performed by the `if` atom.

1.8 Generators

A generator is a routine that can be used to control the iteration behaviour of a loop. For example, this PHP program will print first ten Fibonacci numbers:

```
function fibonacci(int $limit):generator {

  yield $a = $b = $i = 1;
  while (++$i < $limit) {
    $b = $a + $b;
```

```
    yield $b - $a;
    $a = $b;
  }
}
foreach (fibonacci(10) as $n) {
  echo "$n\n";
}
```

This mechanism may be implemented in EO:

```
[limit f] > fibonacci
  memory 0 > a
  memory 0 > b
  memory 0 > i
  seq > @
    a.write 1
    b.write 1
    f 0
    while.
      seq (i.write (i.plus 1)) (i.lt limit
          )
      [idx]
        seq > @
          b.write (a.plus b)
          a.write (b.minus a)
          f (b.minus a)
    TRUE
fibonacci > @
  10
  [n]
    QQ.io.stdout > @
      QQ.txt.sprintf "%d\n" n
```

Here, the generator is turned into an abstract object `fibonacci` with an extra parameter `f`, which is specified with an abstract object argument, which prints what's given as `n`.

1.9 Types and Type Casting

A type system is a logical system comprising a set of rules that assigns a property called a type to the various constructs of a computer program, such as variables, expressions, functions or modules. The main purpose of a type system is to reduce possibilities for bugs in computer programs by defining interfaces between different parts of a computer program, and then checking that the parts have been connected in a consistent way. In this example, a Java method `add` expects an argument of type `Book` and compilation would fail if another type is provided:

```
class Cart {
  private int total;
  void add(Book b) {
    this.total += b.price();
  }
}
```

The restrictions enforced by Java type system in compile time through types `Cart` and `Book` may be represented in EO by means of decorators, for example:

```
[] > original-cart
  memory 0 > total
  [b] > add
    total.write (total.plus (b.price))
[] > cart
  original-cart > @
  [b] > add
    if.
      b.subtype-of (QQ.txt.text "Book")
      @.add b
      []
        error "Type mismatch, Book
            expected" > @
```

Here, it is expected that the parameter `b` is defined in a "class," which has `subtype-of` attribute, which may also be provided by a decorator (a simplified example):

```
[] > original-book
  memory 0 > price
[] > book
  original-book > @
  [t] > subtype-of
    t.eq "Book" > @
  [] > price
    @.price > @
```

This decoration may be simplified through a "fluent" supplementary object:

```
type "Cart" > cart-type
.super-types "Object" "Printable"
.method "add" "Book"
cart-type original-cart > cart
```

Here, the `type` object implements all necessary restrictions Java type system may provide for the type `Cart` and its methods.

Type casting, which is a mechanism of changing an expression from one data type to another, may also be implemented through same decorators.

1.10 Reflection

Reflection is the ability of a process to examine, introspect, and modify its own structure and behavior. In the following Python example the method `hello` of an instance of class `Foo` is not called directly on the object, but is first retrieved through reflection functions `globals` and `getattr`, and then executed:

```
def Foo():
    def hello(self, name):
        print("Hello, %s!" % name)
obj = globals()["Foo"]()
getattr(obj, "hello")("Jeff")
```

It may be implemented in EO just by encapsulating additional meta information in classes explained in Sec. 1.4.

1.10.1 Monkey Patching. Monkey patching is making changes to a module or a class while the program is running. This JavaScript program adds a new method `print` to the object `b` after the object has already been instantiated:

```
function Book(t) { this.title = t; }
var b = new Book("Object Thinking");
b.print = function() {
  console.log(this.title);
}
b.print();
```

This program may be translated to EO, assuming that `b` is being held by an attribute of a larger object, for example `app`:

```
[] > app
  cage 0 > b
  b' > copy
  seq > @
    b.write
      [] > book
        [t] > new
          memory "" > title
          title.write t > @
    copy.<
    b.write
      [] > book
        [t] > new
          seq > @
            []
              copy > @
              [] > print
              QQ.io.stdout (^.title)
    b.print
```

Here, the modification to the object `book` is happening through making a copy of it, creating a decorator, and then storing it to where the original object was located.

1.11 Static Methods

A static method (or static function) is a method defined as a member of an object but is accessible directly from an API object's constructor, rather than from an object instance created via the constructor. For example, this C# class consists of a single static method:

```
class Utils {
  public static int max(int a, int b) {
    if (a > b) return a;
    return b;
  }
}
```

It may be converted to the following EO code, since a static method is nothing else but a "global" function:

```
[a b] > utils-max
  if. > @
    a.gt b
    a
    b
```

1.12 Inheritance

Inheritance is the mechanism of basing a class upon another class retaining similar implementation. In this Java code class `Book` inherits all methods and non-private attributes from class `Item`:

```
class Item {
  private int p;
  int price() { return p; }
}
class Book extends Item {
  int tax() { return price() * 0.1; }
}
```

It may be represented in EO like this:

```
[] > item
  memory 0 > p
  [] > price
    p > @
[] > book
```

```
  item > i
  [] > tax
    (QQ.math.number (i.price)).as-float.
       times 0.1 > @
```

Here, composition is used instead of inheritance.

1.12.1 Prototype-Based Inheritance. So called prototype-based programming uses generalized objects, which can then be cloned and extended. For example, this JavaScript program defines two objects, where `Item` is the parent object and `Book` is the child that inherits the parent through its prototype:

```
function Item(p) { this.price = p; }
function Book(p) {
  Item.call(this, p);
  this.tax = function () {
    return this.price * 0.1;
  }
}
var t = new Book(42).tax();
console.log(t); // prints "4.2"
```

This mechanism of prototype-based inheritance may be translated to the following EO code, using the mechanism of decoration:

```
[p] > item
  memory 0 > price
  [] > new
    price.write p > @
[] > book
  [p] > new
    item p > @
    [] > tax
      times. > @
        as-float.
          QQ.math.number (^.price)
        0.1
QQ.io.stdout
  QQ.txt.sprintf
    "%f"
    tax.
      book.new 42
```

1.12.2 Multiple Inheritance. Multiple inheritance is a feature of some object-oriented computer programming languages in which an object or class can inherit features from more than one parent object or parent class. In this C++ example, the class `Jack` has both `bark` and `listen` methods, inherited from `Dog` and `Friend` respectively:

```
#include <iostream>
class Dog {
  virtual void bark() {
    std::cout << "Bark!";
  }
};
class Friend {
  virtual void listen();
};
class Jack: Dog, Friend {
  void listen() override {
    Dog::bark();
    std::cout << "Listen!";
  }
};
```

It may be represented in EO like this:

```
[] > dog
  [] > bark
    QQ.io.stdout "Bark!" > @
[] > friend
  [] > listen
[] > jack
  dog > d
  friend > f
  [] > listen
    seq > @
      d.bark
      QQ.io.stdout "listen!"
```

Here, inherited methods are explicitly listen as attributes in the object `jack`. This is very close to what would happen in the virtual table of a class `Jack` in C++. The EO object `jack` just makes it explicit.

1.13 Method Overloading

Method overloading is the ability to create multiple functions of the same name with different implementations. Calls to an overloaded function will run a specific implementation of that function appropriate to the context of the call, allowing one function call to perform different tasks depending on context. In this Kotlin program two functions are defined with the same name, while only one of them is called with an integer argument:

```
fun foo(a: Int) {}
fun foo(a: Double) {}
```

```
foo(42)
```

It may be represented in EO like this:

```
[args...] > foo
  (args.at 0) > a0
  if.
    a0.subtype-of "Int"
    first-foo a0
    second-foo a0
foo 42
```

This code expects arguments of `foo` to be equipped with the type system suggested in Sec. 1.9. The attribute `subtype-of` will help dispatching the call to the right objects.

1.14 Java Generics

Generics extend Java's type system to allow a type or method to operate on objects of various types while providing compile-time type safety. For example, this Java class expects another class to be specified as `T` before usage:

```
class Cart<T extends Item> {
  private int total;
  void add(T i) {
    total += i.price();
  }
}
```

It may be represented in EO like this:

```
[] > cart
  memory 0 > total
  [i] > add
    total.write > @
      total.plus (i.price)
```

As the example demonstrates, the presence of generics in class declaration may be ignored, since EO is a language without types and type checking.

1.15 C++ Templates

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types, allowing a function or class to work on many different data types without being rewritten for each one.

```
template<typename T> T max(T a, T b) {
  return a > b ? a : b;
}
int x = max(7, 42);
```

It may be represented in EO like this:

```
[a b] > max
  if. > @
    a.gt b
    a
    b
max 7 42 > x
```

As the example demonstrates, the presence of templates may be ignored, since EO is a language without types and type checking.

1.16 Mixins

A mixin is a class that contains methods for use by other classes without having to be the parent class of those other classes. The following code demonstrates how Ruby module is included into a class:

```
module Timing
  def recent?
    @time - Time.now < 24 * 60 * 60
  end
end
def News
  include Timing
  def initialize(t)
    @time = t
  end
end
n = News.new(Time.now)
n.recent?
```

This code may be represented in EO by just copying the method `recent?` to the object `News` as if it was defined there.

1.17 Annotations

In Java, an annotation is a form of syntactic metadata that can be added to source code; classes, methods, variables, parameters and Java packages may be annotated. Later, annotations may be retrieved through Reflection API. For example, this program analyzes the annotation attached to the class of the provided object and changes the behavior depending on one of its attributes:

```
interface Item {}
@Ship(true) class Book { /*..*/ }
@Ship(false) class Song { /*..*/ }
class Cart {
  void add(Item i) {
    /* add the item to the cart */
```

```
    if (i.getClass()
      .getAnnotation(Ship.class)
      .value()) {
      /* mark the cart as shippable */
    }
  }
}
```

The code may be represented in EO using classes suggested in Sec. 1.4:

```
[] > book
  ship TRUE > a1
  [] > new
    [] > @
[] > song
  ship FALSE > a1
  [] > new
    [] > @
[] > cart
  [i] > add
    # Add the item to the cart
    if. > @
      i.a1.value
        # Mark the cart shippable
        TRUE
        FALSE
```

Annotations become attributes of objects, which represent classes in EO, as explained in Sec. 1.4: `book.a1` and `song.a1`.

2 Traceability

Certain amount of semantic information may be lost during the translation from a more powerful programming language to EO objects, such as, for example, namings, line numbers, comments, etc. Moreover, it may be useful to have an ability to trace EO objects back to original language constructs, which they were motivated by. For example, a simple C function:

```
int f(int x) {
  return 42 / x;
}
```

It may be mapped to the following EO object:

```
[x] > f
  [] > @
    "src/main.c:1-1" > source
    42.div x > @
  "src/main.c:0-2" > source
```

Here, synthetic `source` attribute represents the location in the source code file with the original C code. It's important to make sure during translation that the name `source` doesn't conflict with a possibly similar name in the object.

3 Conclusion

We demonstrated how some language features often present in high-level object-oriented languages, such as Java or C++, may be expressed using objects. We used EO as a destination programming language because of its minimalistic semantics: it doesn't have any language features aside from objects and their composition and decoration mechanisms.

References

Bugayenko, Yegor (2021). *EOLANG and phi-calculus*. arXiv: 2111.13384 [cs.PL].

Ceccato, Mariano, Paolo Tonella, and Cristina Matteotti (2008). "Goto Elimination Strategies in the Migration of Legacy Code to Java". In: *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE. USA: IEEE, pp. 53–62.

Erosa, Ana M and Laurie J Hendren (1994). "Taming Control Flow: A structured approach to eliminating goto statements". In: *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)*. IEEE. USA: IEEE, pp. 229–240.

Pan, Si and R. Geoff Dromey (1996). "A Formal Basis for Removing Goto Statements". In: *The Computer Journal* 39.3, pp. 203–214.

Williams, M. Howard and G Chen (1985). "Restructuring Pascal Programs Containing Goto Statements". In: *The Computer Journal* 28.2, pp. 134–137.