Eolang Technical Report

Cumulative Report of Stages I,II,III IV & V

HSE Team

Research and Development Project
Client: Huawei Technologies Co. Ltd.
№ TC202012080007

Department of Computer Science
National Research University
Higher School of Economics
Moscow
Russia

Contents

Stage 1

Eolang Analysis

1.1 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. This work seeks to analyse the Eolang language, compare to other OOP languages and develop the core features of this new language. Overall goal of the paper: Building software-intensive, high-load, and reliable/robust applications (e.g. cloud services) based on industry level (compact and reliable) programming language.

1.2 OOP Problems Detected

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 [38]. 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.

In addition, OOP styles were considered to ensure the formation of effective compositions of software products. Among the formal approaches are the work of the theoretical models describing the theoretical OOPs [33]. These include, for example, Abadi's work on sigma-calculus, which can be used to reduce the semantics of any object-oriented programming language to four elements: objects, methods, fields, and types. Further development of these works led to the creation of a phi-calculus used in the description of elementary design patterns.

1.2.1 The aim

This work aims to analyze the Eolang language, describe the syntax and semantics of Eolang to assess the quantitative and qualitative characteristics of Eolang through the prism of comparing [46], Eolang with other object-oriented programming languages such as Java and C ++ [45]. Summarize and check compilability and utility in simple cases, making the solution available to the world developers' community.

1.3 Related Works

Kotlin language specification [94], Comparative Studies of Programming Languages [34] , Direct Interpretation of Functional Programs for Debugging [47].

1.4 Solution

We want to stay as close to Java and JVM as possible, mostly in order to re-use the Eco-system and libraries already available. We also want to have an ability to compile it to any other language, like Python, C/C++, Ruby, C-sharp, etc. In other words, EO must be platform independent [40].

1.4.1 Metrics

Before providing test cases we have chosen main metrics according to the current level of EO-compiler and EO-objects at all.

Syntax comparison

We decided to choose this metric to formalize and distinguish Emprical comparison. Rules written in RBNF (like a text in a programming language) consist of separate elements - tokens. Lexemes are the names of concepts that are called nonterminal symbols or simply nonterminals in the theory of formal languages. For example, in the rule StatementSequence = Operator ";" Operator. Nonterminals are OrderOperator and operator . Terminal symbols are the characters that make up the final (terminal - final, final) program. When writing to RBNF, terminal characters are written in quotation marks. In the given example, one terminal is ";". Terminal symbols are also RBNF tokens. Finally, tokens include special characters used in the RBNF itself. In the sequence of statements rule, these are the equal sign, curly braces, and a period at the end.

The total number of tokens in a language syntax description can serve as a generalized characteristic of the size of this description. It is much better to use the number of tokens as a measure of volume than, say, the number of characters in the description. In this case, the value of our criterion will not depend on what language (Russian, English) or what specific words the nonterminals are called - the concept of language. The number of different non - terminals is the next characteristic that we will calculate. The number of concepts used to describe a language is undoubtedly the most important property on which the ease of mastering this language depends. It can be noted that the number of nonterminals must be equal to the number of rules in the syntax description, since exactly one rule must exist for each concept. Set and the number of different terminal symbols of the language mentioned in the syntactic formulas characterize the vocabulary of the language - a set of characters and special words. In all the languages we are discussing (EO, Java, Groovy), there are special words that can be used only in a strictly defined case. The programmer should actually know them by heart. Counting the number of special words will allow you to estimate the volume of programming.

Empirical comparison based on OOP constructs

LOC (lines of code), Readability, Maintainability, Cohesion and coupling,Code smells.

The work is :

1. To form a class-diagram for EO solution (optional).
2. To implement the same program with another language (Java, Groovy, Kotlin):
    - variant is to write an equivalent of EO solution.
    - variant is to write the program using non-efficient from the point of EO constructions(inheritance, type casting, flow control statements, etc.).

Execution Time and Memory Usage comparison

Measure the efficiency of Eolang, despite the fact of inefficient compiling process. We implement some prototypes to identify it. These prototypes are:

- **Pi digits**
- **Binary tree**
- **3 types of sorting algorithms**.

  They are: Bubble sort, Insertion sort, Selection sort.

Software reliability

During our comparison let's try to find out software reliability of solutions that use non-reliable(from EO postulate) constructions and analyze cases in which this problem possibly arises.

The argument for this is that the only proof of non-reliable construction is one that is a procedural approach .

1.4.2 Exploring the Eolang Interoperability with Java

The syntaxes of the above languages are not very different from that of Java so it makes it easy for them to support Java and interoperate. Xtend language compiles to Java but the syntax is rather very close to that of Java. From the above findings one of the major mechanisms enable these languages to interoperate with Java is to share some similarities with Java language and support some Java syntax. Eolang, currently, barely supports or shares any features with Java.

With the current implementation of Eolang, perhaps some core objects could be developed to handle parsing java objects so that programmers can simply create and make copies of Java objects in Eolang. This could be one of the ways Eolang might support Java, since supporting some Java syntax or code is a major mechanism that cuts across all the languages discussed above, in terms of their interoperability. The objects will seem like a wrapper library for Java APIs to make them interoperable. This can be a good start with progressive updates to, perhaps, establish full support of Java and full interoperability of Eolang and Java. The above languages in some sense are like super sets of Java. Thus, they extend more features and also support Java features.

Eolang could increase the number of atoms in the language core library. In Eolang atoms (if, sprintf, add, stdout, and length) have to be implemented in Java, not in Eolang. These objects are datarized at run-time. Although this may provide access to use some Java objects in Eolang, it may not establish full interoperability with Java. But this can be later improved on. Perhaps regular programmers's could be given the chance to be able to make custom atom by writing their own java classes and somehow making them atoms in Eolang. This might be rather a lot of work because every Java object may have to be developed as atoms in Eolang, for use in Eolang.

The ability to support Java syntax is vital for achieving interoperability. The benefit of running on a JVM is that you can take advantage of all the frameworks and tools built into other JVM languages, and byte-codes (.class) can communicate in the JVM.

Eolang may consider providing some core objects that communicate with Java, like kotlin's CallJava and CallKotlin objects [60]. Increasing the range of collection of atoms in Eolang may be a good start towards interoperability with Java.

1.5 Discussion

We faced some issues with some bugs and optimization problems. Especially when we tried to implement new test cases, we needed to implement new functionality and fix bugs that are caused by new implementations. All results are provided by weekly reports and github issue section `https://github.com/cqfn/eo/issues`. The most important one are connected with memory organisation, especially memory attributes `https://github.com/cqfn/eo/issues/248`.

Otherwise the lack of technical documentation about compiler makes the process of development like a Pandora's box, because each new iteration of implementing new features in the object library causes a lot of bugs and errors while we pull (https://github.com/cqfn/eo/pull/185) the description of EO-libraries there is still no technical documentation.

Objectionary, the idea of implementating objectionary has a great potential because we will have more structured EOlang objects library, but at the moment we don't have the complete solution (`https://github.com/yegor256/objectionary`).

The actual version of EO-compiler (`https://github.com/cqfn/eo/issues/250`) is also a topic for discussion now because we need to release new version after each fixed bug to be able to continue the prototypes development.

The performance problem is connected with existence of redundant objects in EO runtime, too much nesting of calls. It is required to make the EO runtime, flatter, linear and simpler.

1.6 Deliverables

1.6.1 Languages

As it was already mentioned we tried to be as close to JVM languages as possible We examine next languages:

Java

The Java programming language originated as part of a research project to develop advanced software for a wide variety of network devices and embedded systems. The goal was to develop a small, reliable, portable, distributed, real-time operating platform. When the project started, C++ was the language of choice. But over time the difficulties encountered with C++ grew to the point where the problems could best be addressed by creating an entirely new language platform. Design and architecture decisions drew from a variety of languages such as Eiffel, SmallTalk, Objective C, and Cedar/Mesa. The result is a language platform that has proven ideal for developing secure, distributed, network-based end-user applications in environments ranging from network-embedded devices to the World-Wide Web and the desktop.

Kotlin

Kotlin was designed to run on the JVM. It comes with its own compiler that compiles kotlin code to byte-code that can run on the JVM. The byte-code generated by kotlin compiler is equivalent to the byte-code generated by the Java compiler. Whenever two-byte code files run on JVM, due to their equivalent nature they can communicate with each other and that's how interoperability is established in kotlin for Java . Kotlin was developed keeping interoperability in mind. A Kotlin class or a function can refer to the Java classes and their methods in a simple way. Kotlin program files (.kt) and Java program files (.java) can be in the same project. They are all compiled and converted to .class files which are byte-codes.

Groovy

Groovy is a dynamic and optionally typed object oriented scripting language. Just like Kotlin and Scala, groovy is interoperates well with Java; almost all Java code are also valid Groovy code.

1.6.2 Abstraction

EO

Already exists. The object abstracts behavior (or cohesion) of other objects.
LOC:

```
[x a b c] > polynomial
  (((x.pow 2).mul a).add (x.mul b)).add c > @
```

Java

To create an abstract class, just use the abstract keyword before the class keyword, in the class declaration.
LOC

```
class Polynomial {
    public Polynomial(int x, int a, int b, int c){
    this.x=x;
    this.a=a;
    this.b=b;
    this.c=c;
    }
    public int compute() {
        return x*x*a + x*b + c;}
```

```
    }
}
```

Groovy

To create an abstract class, just use the abstract keyword before the class keyword, in the class declaration.
LOC

```
class Polynomial {
    public Polynomial(int x, int a, int b, int c){
    this.x=x;
    this.a=a;
    this.b=b;
    this.c=c;
    }

public int compute() {
    return x*x*a + x*b + c;}
    }
```

Kotlin

Like Java, abstract keyword is used to declare abstract classes in Kotlin. An abstract class cannot be instantiated (you cannot create objects of an abstract class). However, you can inherit subclasses from can them.

1.6.3 Encapsulation

EO

Doesn't exist and will not be introduced

Java

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

Groovy

In Groovy, everything is public. There's no idea of private fields or methods. At least, not in the way they would be represented in C++ or Java.

Kotlin

OOP encapsulation in Kotlin unlike Python is enforced and has some fine grained levels (scope modifiers/keywords)

1.6.4 Inheritance

EO

Doesn't exist and won't be introduced. The usual inheritance is presented by decorators (@) The main idea is that in the production (inheritance couses many problems) `https://www.yegor256.com/2016/09/13/inheritance-is-procedural.html`

Java

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- sub-class (child) - the class that inherits from another class.
- super-class (parent) - the class being inherited from to inherit from a class, use the extends keyword.

Groovy

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order. The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as super-class (base class, parent class). "extends" is the keyword used to inherit the properties of a class.

Kotlin

All classes in Kotlin have a common super-class Any, that is the default super-class for a class with no super-types declared. Any has three methods: equals(), hashCode() and toString(). Thus, they are defined for all Kotlin classes. By default, Kotlin classes are final: they can't be inherited. To make a class inheritable, mark it with the open keyword.

1.6.5 Polymorphism

EO

Will be implemented (Ad hoc polymorphism)

Duck typing in computer programming is an application of the duck test — "If it walks like a duck and it quacks like a duck, then it must be a duck " — to determine whether an object can be used for a particular purpose. With normal typing, suitability is determined by an object's type. In duck typing, an object's suitability is determined by the presence of certain methods and properties, rather than the type of the object itself.

Java

Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

Groovy

If you do this in Java (splitting the classes out to their own files of course), it won't compile. Java looks at the reference type for available methods, so you will get a NoSuchMethodException. In Groovy, however, it looks at the type of the object, not the type of the reference so the method is found at runtime.

Kotlin

Kotlin supports two forms of polymorphism because it is both strongly and statically typed. The first form of polymorphism happens when the code is compiled. The other form happens at runtime. Understanding both forms of polymorphism is critical when writing code in Kotlin.

1.6.6 Data types

EO

Presented like Atom Data Type `https://stackoverflow.com/questions/10525511/what-is-the-atom-data-ty`
It is an acronym of "Access to Memory". It is a term used for simple numerical identifiers (other name is "handles") which represent some internal data structures in the system.

Moreover,object modeling/representation is used in different programming languages (Smalltalk, Oberon, Zonnon etc.). Function as an object model is commonly used in functional programming languages. Type inference in programming languages helps to deduce types

Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

Primitive data types: The primitive data types include Boolean, char, byte, short, int, long, float and double. Non-primitive data types: The non-primitive data types include Classes, Interfaces and Arrays.

Groovy

Groovy supports the same number of primitive types as Java.

Kotlin

In Kotlin, everything is an object in the sense that we can call member functions and properties on any variable. Some of the types can have a special internal representation - for example, numbers, characters and boo leans can be represented as primitive values at runtime - but to the user they look like ordinary classes.

1.6.7 Performance results

According to the provided tests we underline that the results are under pressure, because of lack of optimization. Especially we faced up with a problem of StackOverflowError while trying to compile hard tests(100,1000..etc. args).

Factorial of 6

```
[n] > factorial
 if. > @
   n.eq 0
   1
   n.mul (factorial (n.sub 1))
```

Program's consumption at runtime: 236 MB

Maximum of an array of 8 nums

```
[args...] > max
  [accumulator current] > reduceFunction
    if. > @
      current.toInt.less accumulator
      accumulator
      current.toInt
  reduce. > biggest
    args
    0
    reduceFunction
  stdout > @
    sprintf
      "%d\n"
      biggest
```

Program's consumption at runtime: 250 MB

Linear search for an array of 5 numbers. The last argument passed in the command line is the search key and the elements before it makes the array

.

```
[args...] > max
  memory > key
  memory > temp
  (args.get 5).toInt > elem
  memory > x
  seq > @
    key.write -1
    x.write 0
    while.
      x.less 5
      [i]
        seq > @
          temp.write (args.get i).toInt
          (elem.eq temp).if
            key.write i
            false
          x.write (x.add 1)
    stdout
      sprintf "Element index: %d\n" key
```

Program's consumption at runtime: 193 MB

1.7 Conclusion

1.7.1 Overall

While in the development of complex software systems, universal languages that support the multi-paradigm style are often used, there are extensive classes of problems that are well suited to OOP languages with increased reliability. As part of the work, the ways of creating such a language are considered. It has been shown that several unreliable designs are often unnecessary. Some of the presented designs can be replaced with more reliable ones.

Theoretically promising direction of development of languages with objects is a functional approach, in which the function acts as an object, and its arguments - as attributes and/or methods. In this case, it is possible to provide some neutrality both in terms of typing and in terms of function parameters representing attributes and methods. Additional benefits are associated with parametric polymorphism, type inference and computational strategies - call-by-name, call-by-value, call-by-need, etc. To a certain extent these mechanisms are implemented in the languages LISP, (S) ML, Miranda [5], Haskell [20]. The multi-paradigm approach takes place in the programming language F-sharp [44]. It is possible to "immerse" fragments of the source code written in a wide range of languages into an abstract/virtual machine (such as CLR or JVM) that broadcasts components into intermediate code (MSIL or byte) respectively. As we have described above, existing OOP languages exhibit high complexities and many design flaws.

Positive results are connected with compact and clear style of language syntax and the LOC.

While the opposite EO-Language is still in the process of improving compiler (e.g. not all lists of important objects are implemented) and even its concepts (terminology and theory).

For now there is a risk on the development of large-scale programs because we only implemented algorithmic tasks (`https://github.com/HSE-Eolang`), but with the current architecture of Eolang, large-scale programs might not be easy/possible to implement.

One more conclusion is connected to further research on Eolang interoperability, not only with Java, but with other JVM languages too (Kotlin, Scala, Groovy).

1.7.2 Article comments

As a part of our 1 part of the research we have some comments according to the EO-article by [96],where the concept of language is produced.

Abstract objects appear to resemble classes in their essence. That is, the impression of a substitution of terminology is created. So why not use the concept of a class instead of an abstract object?

It's not clear about polymorphism. Typically, in an OOP approach, polymorphism is defined through the principle of substitution, where a super-class or subclass can be replaced with another subclass. That is, we are talking about dynamic binding at runtime. Traditionally, this is done through inheritance and virtualization [59]. The basic idea is that the type of the inline object is unknown at compile time. Only the interface of the super-class (parent class) is known. The description provided does not reflect how child objects are formed to support dynamic polymorphism.It would be interesting to receive more complete information on this issue in order, for example, to understand how to solve the problem of a container containing various geometric shapes.

It can also be noted that there is static polymorphism and dynamic. The text is about static, which is not very interesting. It often happens that objects are formed during computations, when their type cannot be determined by the compiler.

A method is essentially a function that maps an argument to a result [16]. Unlike data, which in the traditional sense is a store of a set value. The attribute in the current understanding is both. In this regard, an object with many attributes must either display them without changes, or change in accordance with the functional transformation. It somehow has to be defined more specifically when defining a given term. Another option is to represent the function as an object, which essentially means a different name. Something similar is used in C ++ functors when parentheses are overridden. Finally, the absence of types does not allow you to uniquely identify free attributes during compilation, which leads to errors in the use of free attributes. It is impossible to identify objects that are passed as parameters, there is no way to pass objects as parameters and there is no type casting.

Stage 2

Eolang Compiler For Simple Cases

2.1 Abstract

Eolang is an Object-Oriented Programming language aimed at realizing the pure concept of object-oriented programming, in which all components of a program are objects. Eolang's main goal is to prove that fully object-oriented programming is possible not only in books and abstract examples but also in real program code aimed at solving practical problems. The EO programming language is a research and development project that remains in an undeveloped state. Our main objective is to develop the language with more reliable programming methods. In this work, we show our ideas and solution plan and our current progress in achieving this objective.

2.2 Introduction

Yegor Bugayenko, the author of the "Elegant Objects" concept underlying the EO language, identifies the following problems of existing Object-Oriented Programming languages [50] :

1. Mutability of data generates side effects, which leads to unpredictable behavior of the program.
2. Using null references to denote the absence of a value.
3. Applying reflection breaks the encapsulation of classes, allowing the developer to use classes in unpredictable scenarios.
4. Inheritance allows you to get rid of visible duplication of code in similar classes, however, making any changes to parent classes only becomes more complicated because of the need for consistency of changes with all inherited classes.

Eolang is designed to curb the above points. This work aims to contribute to the development of the Eolang compiler that will be accessible at GitHub repository (OSS/MIT licence). It is expected to be able to build java classes from Eolang source code. The maven wrapper plugin will be used to initiate compiling at build phase to generate class files, and then packaged by Maven later. The result of such a solution will bring us to answer the question: Is it possible to rewrite Java Cloud web-module with Eolang?

2.3 Solution Approach

2.3.1 Why it was important to reform the previous version of compiler

The previous version of the compiler was synthetic and just a text translation of EO. It only simulated the behavior of objects in EO with poor performance and had many objects in memory. Above all, it showed potential difficulty in scalability (e.g., add typing) in case of future development. To address these issues, a new model is proposed for the compiler development.

2.3.2 Development plan

The development of the compiler is divided into separate parts and covers the following features:

1. Translation of Eolang program code into ".java" files in the Java programming language.
2. Programming languages used in the implementation of the source code of the translator, thus, objects of the standard library, and the runtime library - Java.
3. The parser (as well as the lexer) must be implemented (and augmented) as part of the ANTLR4 solution.
4. The translation of the output code in Java is done through the construction and processing of an intermediate model obtained through parsing XML documents of the stage of parsing a program in EO.
5. The final solution (translator) should be delivered in an easy-to-use form: a Maven plugin [30] in the Maven Central repository [63].

2.3.3 Eolang Objects Translation

This section describes how the translation of the user-defined Eolang object will be performed.

Naming & Scoping Rules

An Eolang object named "obj" of the global scope is translated into a Java class named "EOobj.java". The prefix "EO" is used as a simple measure to escape naming conflicts with Java code. Eolang objects of local scopes (bound attributes of other objects or anonymous objects passed as arguments during application) are not present as separate Java files. Instead, these are put to the scopes they originate from. So, attributes are named private inner classes, and anonymous objects are anonymous Java classes. This technique makes target code denser and delegates the management of scopes and child-parent hierarchies to the Java Virtual Machine (JVM), not the Eolang runtime. The source program "obj" object is translated into a single "EOobj" class in a single EOobj.java file. No manual naming and class connections management. The transpiler is to compile the source program to the target platform as it is organized originally and let the Java Virtual Machine deal with all the relationships,hierarchies, and scoping mechanisms locally (native).

Target Class Hierarchy

The target class extends org.eolang.core.EOObject.

Constructors

The target class has only one constructor for simplicity. There are two possible cases:

1. If the source class has no free attributes, the resulting constructor has no arguments.
2. If the source class has free attributes, the resulting constructor has all the arguments in the order specified in the source program. All arguments of the constructor are of type "EOObject", so the object does not know the actual type. In the case where free attributes are present, the default constructor (in Java) is disabled.

EO Objects Free Attributes Translation

Free attributes of the source object are translated as follows. Assuming there are these free attributes in the source Eolang object (in the order specified):

1. a
2. b
3. c
4. class
5. he133

For each of them, the following will be performed:

1. Generation of a private final field named EO<attributeName>. ”EO” prefix is used to allow using some special names from the source language (e.g., class) that may be a reserve word in the target platform (Java). The field is of type EOObject.

2. Generation of a public method named EO<name>. The method's return type is EOObject. The method's body is just return this.EOattr;.

3. The body of the constructor sets the private EO<attributeName> field to the value of the argument EOattr. The proposed model is fully immutable since a code fragment that applies (or copies) a class can set the free attributes only once through the constructor. Fields holding free attributes are private and final. The default (empty) constructor is disabled. Once set, an attribute cannot be changed either from a user code fragment or from the inside of a class itself.Hence, objects are fully immutable.

However, the model has some weaknesses. First, it cannot handle the partial application mechanism. The proposed solution allows the “fully-applied-at-once” copying technique only.

Secondly, just as in the original transpiler, the model does not perform type checking of the objects being passed for free attributes binding.

Eolang Objects Bound Attributes Translation

Bound attributes of the source object are translated as follows.

1. For every bound attribute with an arbitrary name such as ”attr”, a method named ”EOattr” is generated. The method returns an object of type EOObject.

2. If the bound attribute is constructed through application operation in the source program, then the target code is placed into the method ”EOattr” being generated. In this case the target code is generated as follows:

   (a) The method returns a new instance of the object being applied in the source program.

   (b) The instance is created through its constructor, where all its parameters are passed in the order of the order specified in the source program.

   (c) The evaluation strategy is down to the instance being returned (it decides how to evaluate itself on its own).

3. If the bound attribute is constructed through abstraction operation in the source program, then a private inner class named ”EOattr” is generated. The inner class is a subclass of the ”EOObject” base class. The target code is generated as follows in this case:

   (a) Free attributes of the abstracted object are translated as described in subsection ”EO Objects Free Attributes Translation”.

   (b) Bound attributes of the abstracted object are translated as described in this section.

   (c) The method ”EOattr” that wraps the ”EOattr” private inner class returns a new instance of that class passing all arguments (free attributes) to it in the order specified according to the source program.

   (d) The evaluation strategy is implemented via overriding the ”getDecoratedObject” standard method (as described in detail in the Dataization section).

   (e) To access the parent object, the ”getParent” is used from EOObject (it is also overridden inside nested classes). It is done for simplicity and may change in the distant future when the compiler [25] becomes more feature-rich.

   (f) When the source program explicitly accesses an attribute (e.g., EOattr) of the parent object (e.g., EOobj) of the attribute (e.g., EOboundAttr), it is explicitly translated to “EOobj.this.EOattr”.

It is important to mention that memory leakage issues that inner classes are blamed for do not affect the proposed model performance compared to the original implementation of the compiler (current compiler) since the latter links all the nested objects (attributes) to their parents. As a further optimization, the transpiler may check if a bound attribute created through abstraction does not rely on the parent's attributes. In this case, the private nested class may be made static

(i.e. not having a reference to its enclosing class). As it was mentioned in Naming & Scoping Rules, bound attributes (these are basically nested objects) are put to the scopes they originate from. So, bound attributes are translated into named private inner classes. This technique makes target code denser and also delegates the management of scopes and child-parent hierarchies to the Java Virtual Machine (JVM), not the Eolang runtime.

Anonymous Objects

The idea behind anonymous objects is simple. Every time an anonymous object is used in the source program (either in application to another object or in decoration), it is translated as an local class of the EOObject class right in the context it is originated from. All the principles of class construction take place in the local class just as in named classes (constructor generation, free and bound attributes translation, implementation of decoration & dataization mechanism, etc.).

Decoration & Dataization

Every user-defined object evaluation technique is implemented via an overridden "getDecoratedObject" standard method. The overridden method constructs the object's decoratee and delegates the object's own evaluation to the decoratee. The result of the evaluation of the decoratee is then returned as the result of the evaluation of the object itself.

If the decoratee is not present for the source program object, an exception is thrown inside the dataization getData method

Accessing Attributes in the Pseudo-Typed Environment

If the user code fragment utilizes an EOObject instance (and the concrete type is unknown), the Java Reflection API is used to access the attributes of the actual subtype. Since both free (inputs) and bound (outputs) attributes are wrapped as methods (that can have from zero to an arbitrary number of arguments) it is possible to handle access to all kinds of attributes through a uniform technique based on the dynamic method invocation Java Reflection API. The technique works as follows:

1. Ask Reflection API to check if a method with the name EOattr and the necessary signature (with the correct number of arguments) exists. If it does not exist, throw an exception. Otherwise, proceed to step 2.

1.1 If method not found, try to complete Point 1 for the getDecoratedObject().

2. Invoke method passing all the arguments into it in the order specified.

Data Primitives and Runtime Objects

Data primitives and runtime objects also extend the EOObject base class. However, the main idea behind them is to make them as efficient as possible. To do that, data primitives objects are implemented in a more simple way compared to the guidelines of the proposed model. For example, bound attributes of the data primitives objects are implemented through methods only (no class nesting is used).

Another important property of the runtime objects is that they are often atomic and, hence define their evaluation technique without a base on the decoratee as mentioned above. Instead, these dataize to the atomic data or perform a more efficient (semi-eager or eager) evaluation strategy.

2.3.4 Transformation from XML to Medium (intermediate) model

Some dependencies in the XML files of old version of compiler were noted:

1. All abstractions, no matter their true location in the source "*.eo" file, are moved to the first nesting level. The translator makes the tree flat in terms of abstractions. In this way the translator assigns a name to all abstractions (including anonymous ones).

2. All kinds of bound attributes (i.e. abstraction-based & application-based bound attributes) within EO objects are essentially replaced by application in all cases. So, the application remains as application. And abstraction-based attributes turn into an application that refers to a previously nested object on the first nesting level.

Based on these two properties of the XML input document, it was decided to translate from XML to Medium as follows:

1. Translate all abstractions. They are the first level of XML document nesting. Recursion is not applied at this stage, as abstractions are represented in a completely flat (linear) form.

2. For each abstraction from step 1, all internal applications (i.e. bound attributes) are translated. This process is recursive because of the recursiveness of subtrees that belong to the appendices. After step 2, we have a list (array) of abstractions in the Medium model, each of which has a list of bound-attributes defined by applications, which are also translated into the Medium model.

After that several optimizations are made by the resulting code model structure:

1. Code Model Deflate Operation. This operation returns abstractions, which are "flat" (linear) in previous compiler model, to their places. Thus, the applications which defined abstraction-based attributes get a new field (wrappedAbstraction). Also the applications that referenced local abstraction-based objects get this field.

2. Code Model Scope Correction. This operation corrects the Scope of all the Medium model entities. All applications have file, thus, the abstraction object in which they reside. All abstractions have Scope, thus, the parent abstraction object.The operations outlined above help the intermediate Medium code model become closer to the final model, and move away from the XML model. Both optimizations greatly simplify the final code translation from Medium model to Java.

The google java format [48] and Picocog [56] libraries are taken into consideration in order to deliver cleaner and maintainable code. Also ideas from kotlin [54] will be looked at.

2.4 Deliverable

EO-lang compiler accessible at Github repository (OSS, MIT), which can build java classes from Eolang source code. The maven wrapper plugin will be used to initiate compiling at the build phase to generate class files, and then packaged with Maven later.

2.4.1 Runtime Model

Class	Description
EOObject	1.Instantiate an EO object (i.e. make a copy of it via a constructor). 2.Dataize an EO object 3.Set a parent object 4.Access attributes of an object
EOData	Used to store data primitives
EODataObject	A wrapper class to interpret EOData as EOObject
EONoData	A primitive class representing the absence of data

2.4.2 Mapping EO Entities to Java Code Structures

EO	Java
Abstraction	A plain old Java class extending EOObject. The plain old Java class may also be nested or local.
Application	1.Creating a new class instance 2. Direct method call 3.Calling the method via recursion
Package-scope application	Is wrapped by Abstraction because of the absence of package-levels in Java
A free attribute	A class field that should be set via the constructor
A bound attribute	A method that returns an object
Duck typing	Everything is EOObject. Class fields are accessed via Java Reflection
Object dataization	It is the default for all user-defined classes and relies on the getDecoratedObject()
getDecoratedObject()	To be overridden for evaluation strategy and used to access the attributes of the decoratee object
Anonymous EO objects	Implemented via Local Java Classes Nested class — attribute Local class — anonymous object

2.4.3 Examples

Sample EO Program (covers almost all language features)

:

```
[a b] > rectangle
  a.mul b > area
  [] > perimeter
    2 > a
    mul. > @
      a
      add.
        ^.a
        ^.b
```

0-th level global object rectangle

```
<o line="5" name="rectangle"
                  original-name="rectangle">
 <o line="5" name="a"/>
   <o line="5" name="b"/>
   <o base=".mul" line="6" method="" name="area">
      <o base="a" line="6" ref="5"/>
      <o base="b" line="6" ref="5"/>
   </o>
   <o base="rectangle$perimeter"
      cut="5"
      line="7"
      name="perimeter"
      ref="7">
      <o as="a" base="a" level="1" ref="5"/>
```

```
      <o as="b" base="b" level="1" ref="5"/>
      <o as="area" base="area" level="1" ref="6"/>
   </o>
</o>
```

1. free attribute a
2. free attribute b
3. application-style bound attribute area
4. abstraction-style bound attribute perimeter

2.4.4 Tests (Execution Time)

Fibonacci

50-th Fibonacci number [85]:
12586269025
Time: 0m0,191s

Pi digits

Pi Digits till 400000 numbers after dot [88]: 3.141595..
Time: 0m4.023s

Factorial

Factorial of 20 [84]:
2432902008176640000
Time: 0m0.126s

Sum of array elements

First 50 elements
(1-50) sum [90]: 1275
Time: 0m0.179s

2.5 Discussion

2.5.1 Limitation

Anonymous objects must have no free attributes

In a situation where an object needs to be pass to, for instance, an array.map, it is expected to create an object that has an attribute with output parameters, as opposed to that of the previous runtime model.

This is due to the fact that an instance of the class cannot be created without passing the expected arguments. This happens where anonymous objects are used. In future, it may be possible to resolve this issue at the translator level so that the user code will not be affected by this limitation

Recursion now builds only in a lazy mode

What is impossible now: Use recursive definition inside a greedy construct. Greedy constructs include primitive methods like EOadd, EOmul, etc.

Example:

```
2.mul (recursive (n.sub 1))
```

It impossible now, because the recursion, due to greediness, will never be reduced to the result.

What is possible now: Use greed inside recursive constructs

Example:

```
recursive (n.sub 1) (n.mul 2)
```

2.5.2 Possibilities for future development of the model

Now, with an intermediate code model in Java (i.e., an object tree built on the Medium model in Java), we can:

Add call optimization

There is a predictable and an unpredictable context.

1. Predictable:

   ```
   rectangle > rec
   ```

   In this case, rec is a type-predictable variable. it is obvious that this is a type of some.package.name.here.EOrectangle. In a predictable context, we can opt out of recursion. Then

   ```
   rec. attr 5
   ```

   instead of rec.

   ```
   _getAttribute ("attr", EOint(5L))
   ```

   will be translated to

   ```
   rec.EOattr(EOint(5L))
   ```

   An increase in speed and even greater connectivity of objects is achieved. As a bonus: partial validation of the program by the Java compiler is in predictable contexts.

2. Unpredictable context:

   ```
   [a b] > rectangle
   ```

   Here it is not obvious what [a] is. In unpredictable contexts, recursion is continually used. The unpredictable context is still one-access to the input attributes from inside the object. All other contexts are predictable

Add other possible optimization.

If it's possible to simplify some object. Example:

```
[] > obj
    [] > @
        [] > @
            [] > @
                5.add 7 > @
```

The above can be simplify to:

```
5. add 7 > obj
```

Make the translator detect recursive call cases

1. It will try to replace normal recursion from object recursion (when objects are created recursively) to functional recursion (method chain). This will work faster and more efficiently but could also take a long time and resources.
2. Tail recursion, the translator does not only covert into a functional chain, but also converts into a loop. Then the tail recursion in EO will be equivalent to a normal loop but that is even a more complex task than point 1.

2.6 Conclusion

We have presented the solution approach and the progress. As described in the deliverable section, the product of this work is the new Eolang compiler which is now ready and available at GitHub [62]. The new model (transpiler) is embedded into the pipeline of the maven wrapper plugin. The maven wrapper plugin [41] is used to initiate compiling at the build phase to generate class files from Eolang source code, which are subsequently packaged. As displayed above, the performance of this new model shows a great improvement. Appendix (A.1) shows a sample of generated sources by the new runtime for the fibonacci code.

The proposed solution uses Java instead of XSLT[32] because XSLT is not scalable, too complex to comprehend, and can only transform one text to another or one xml fragment to another[31]. A scalable solution should, perhaps, have code that is clearly analyzable.

In the next phase of this research and development, based on the proposed solution, we are going to develop the Java module from cloudBU by rewriting it in Eolang. During this development we will be able to identify bugs, fix them, and add more features to improve the compiler.

Stage 3

Rewriting Java Module in CloudBU in Eolang

3.1 Abstract

EO programming language is a research and development project that is still in progress. In the effort to move it forward, we suggest rewriting Java CloudBU module in EO while finding and fixing bugs along the way. The main objective is to improve the language reliability. In this paper, we explore the Jpeek project on cohesion metrics calculations, and then rewrite them in EO as a CloudBU module.

3.2 Introduction

Software developers aim for systems with high cohesion and low coupling [6]. Cohesion metrics indicate how well the methods of a class are related to each other. Metrics estimate the quality of different aspects of software. In particular, cohesion metrics indicate how well the parts of a system operate together. Cohesion is one of software attributes representing the degree to which the components are functionally connected within a software module [2, 4, 3]. Class cohesion is one of the important quality factors in object-oriented programming because it indicates the quality of class design. There are several metrics to measure class cohesion. typically, a cohesive class performs a single function. A non-cohesive class performs two or more unrelated functions. A non-cohesive class may need to be split into two or more smaller classes. The assumption behind these cohesion metrics is that methods are related if they work on the same class-level variables. Methods are unrelated if they work on different variables altogether. Thus, in a cohesive class, methods work with the same set of variables. In a non-cohesive class, there are some methods that work on different data [108].

A cohesive class performs exactly one function. Lack of cohesion means that a class performs more than one function. This is not desirable; if a class performs any unrelated functions, it should be split up. In summary:

1. High cohesion is desirable since it promotes encapsulation. As a drawback, a highly cohesive class has high coupling between the methods of the class, which in turn indicates high testing effort for that class.

2. Low cohesion indicates inappropriate design and high complexity. It has also been found to indicate a high likelihood of errors. The class should probably be split into two or more smaller classes.

JPeek application is a tool that is designed to analyze code quality by measuring cohesion based on certain metrics. This paper focuses on re-implementing these metrics in the EO programming language, a newly developed Object-Oriented Programming language based on the concept of Elegant Objects.

3.3 The EO-jPeek Solution

The proposed solution is to integrate the metric(s) implemented in EO into the JPeek tool (i.e., a module in EO built into jPeek Repo).

3.3.1 Outline

The proposed CloudBU module is the implementation of some of the metrics of JPeek entirely in EO. This module calculates certain JPeek metrics utilizing the EO programming language and the capabilities of its standard object library. An infrastructure of Java classes integrates EO metrics into the general architecture of JPeek tool. This paper focuses on the general architecture of the proposed module, which includes the following elements:

1. A subroutine that parses and reconstructs the data of a class being assessed in the form of EO objects.
2. A class that integrates EO metrics into the general pipeline of the application, providing input and formatting output for each metric run.
3. A technique that embeds the EO programming language transpiler into the building process of JPeek.

3.3.2 Integrating EO metrics into jPeek pipeline

To isolate EO metrics from XML, we apply the following techniques:

1. Firstly, the *Skeleton.xml* document containing information on the structure details of the class being assessed is parsed and reconstructed in the form of EO objects describing the structure of the class. This stage is done by the *EOSkeleton* class [82]. To call and use EO objects, the *EOSkeleton* class utilizes a basic late binding technique made possible through specific embedding of the EO transpiler into the building process described in section 3.3.3.
2. Secondly, the *EOCalculus* [66] class extracts the "class" EO object from the formed *EOSkeleton* instance and loads it into the metric written in EO. To call EO objects of the metrics, the *EOCalculus* class utilizes a basic late binding technique through specific embeddings of the EO transpiler into the building process described in section 3.3.3. The output of the metric is written to the XML to embed the results to the pipeline of the JPeek tool. This is done in the *EOCalculus* class where we use the following packages to help build and format the output into xml and return the result to the calling method (in *org.jpeek.App* class) of JPeek to include it in the pipeline of JPeek tool:
   (a) *org.xembly.Directives*
   (b) *org.xembly.Xembler*
   (c) *org.jpeek.Header*
   (d) *org.cactoos.collection*
   (e) *com.jcabi.xml*

3.3.3 Embedding the EO transpiler into the building process

Since the EO to Java transpiler [65] enhanced by the Team is distributed as a Maven artifact, it was embedded into the building process as follows:

1. The artifact *org.eolang.eo-runtime* was added as a runtime dependency to the *pom.xml* of the JPeek project. It was needed to access the standard object library of EO at runtime.
2. The artifact *org.eolang.eo-maven-plugin* was added as a plugin as the first step of the building process. The plugin transpiles the EO metrics before building Java classes of the rest of the JPeek project structure. This allows late binding of Java2EO references in Java code and, hence, provides basic variant of Java2EO interoperability.

3.4 Metrics

Fig. 1 shows the metrics tested using a github library call Node Packages [95]. Go to `https://hse-eolang.github.io/` to browse through the results.

Overall score[1] is **4.53** out of 10. Here is the matrix.

4.5312

Metric	E/C[2]	μ[3]	σ[3]	Min	Max	Green[4]	Yellow[4]	Red[4]	Score[5]	Defects[6]	Graph
EO_CAMC	9/37 0%	0.28	0.10	0.20	0.50	0 0%	1 3%	8 22%	0.72 -84%		11%
EO_LCC	4/37 0%	0.16	0.08	0.10	0.30	0 0%	0 0%	4 11%	0.50 -89%		25%
EO_LCOM1	8/37 0%	8.67	5.79	1.00	15.00	8 22%	0 0%	0 0%	10.00 121%		67%
EO_LCOM2	7/37 0%	0.65	0.16	0.40	0.87	4 11%	3 8%	0 0%	6.79 50%		43%
EO_LCOM3	7/37 0%	0.86	0.21	0.50	1.17	6 16%	1 3%	0 0%	8.93 97%		29%
EO_LCOM4	33/37 1%	1.48	1.02	1.00	5.00	33 89%	0 0%	0 0%	10.00 121%		12%
EO_LCOM5	5/37 0%	1.00	0.00	1.00	1.00	5 14%	0 0%	0 0%	10.00 121%		0%
EO_NHD	8/37 0%	0.48	0.20	0.00	0.62	1 3%	5 14%	2 5%	2.94 -35%		12%
EO_OCC	4/37 0%	0.33	0.10	0.25	0.50	0 0%	1 3%	3 8%	1.00 -78%		25%
EO_PCC	1/37 0%	0.43	0.00	0.43	0.43	0 0%	1 3%	0 0%	2.50 -45%		0%
EO_SCOM	4/37 0%	0.04	0.04	0.01	0.10	0 0%	0 0%	4 11%	0.50 -89%		25%
EO_TCC	4/37 0%	0.16	0.08	0.10	0.30	0 0%	0 0%	4 11%	0.50 -89%		25%

The average mistake of individual scores: 85%, average defects rate: 23%.

Figure 3.1: Metrics results on Node Packages [95]

3.4.1 LCOM

LCOM was introduced in the Chidamber and Kemerer metrics suite [108]. It's also called LCOM1 or LOCOM, and it's calculated as follows: Let m be the number of methods, a be the number of attributes and μ_j be the amount of methods, which use attribute j, then

$$LCOM1 = \frac{1}{(1-m)}(\frac{1}{a}\sum_{j=1}^{a}\mu_j) - m$$

LCOM1 = 0 indicates a cohesive class. LCOM1 > 0 indicates that the class should be split into two or more classes, since its variables belong to disjoint sets.

Classes with a high LCOM1 have been found to be fault prone.

A high LCOM1 value indicates disparities in the functionality provided by the class. This metric can be used to identify classes that intend to achieve many different objectives, and consequently are likely to behave in less predictable ways than the classes that have lower LCOM1 values. Such classes could be more error prone to test and split into two or more classes with a better behavior. The LCOM1 metric can be used by senior designers and project managers as a relatively simple way to track whether the cohesion principle is adhered to in the design of an application and advise changes. The implementation of this metric is available at [74, 73].

This implementation has been tested on:

- Node Packages
  See graph here EO-LCOM1

3.4.2 LCOM2 and LCOM3

To overcome the problems of LCOM1, LCOM2 and LCOM3 were proposed.

A low value of LCOM2 or LCOM3 indicates high cohesion and a well-designed class. It is likely that the system has good class subdivision implying simplicity and high re-usability. A cohesive class will tend to provide a high degree of encapsulation. A higher value of LCOM2 or LCOM3 indicates decreased encapsulation and increased complexity, thereby increasing the likelihood of errors.

The choice of LCOM 2 or LCOM3 is often a matter of taste as these metrics are similar. LCOM3 varies in the range [0,1] while LCOM2 is in the range [0,2]. LCOM2>=1 indicates a very problematic class. LCOM3 has no single threshold value.

It is a good idea to remove any dead, variables before interpreting the values of LCOM2 or LCOM3. Dead variables can lead to high values of LCOM2 and LCOM3, thus leading to wrong interpretations of what should be done.

In a typical class whose methods access the class's own variables, LCOM3 varies between 0 (high cohesion) and 1 (no cohesion). When LCOM3=0, each method accesses all variables. This indicates the highest possible cohesion. LCOM3=1 indicates extreme lack of cohesion. In this case, the class should be split. When there are variables that are not accessed by any of the class's methods, 1 < LCOM3 <= 2. This happens if the variables are dead or they are only accessed outside the class. Both cases represent a design flaw. The class is a candidate for rewriting as a module. Alternatively, the class variables should be encapsulated with accessor methods or properties. There may also be some dead variables to remove. If a class has one method or no methods, LCOM3 is undefined. If there are no variables in a class, LCOM3 is undefined. An undefined LCOM3 is displayed as zero.

Parameters of LCOM2 and LCOM3:

- m = number of procedures (methods) in a class
- a = number of variables (attributes) in a class
- mA = number of methods that access a variable (attribute)
- sum(mA) = sum of mA over the attributes of a class

Implementation details
Parameter m is equal to *WMC*. Parameter a contains all variables whether these are Shared or not. Each access to a variable is counted.

$$LCOM2 = 1 - sum(mA)/(m * a)$$

LCOM2 equals the percentage of methods that do not access a specified attribute, averaged over all attributes in the class. If the number of methods or attributes is zero, LCOM2 is undefined and displayed as zero.

$$LCOM3 = (m - sum(mA)/a)/(m - 1)$$

LCOM3 varies between 0 and 2. Values 1...2 are considered alarming.

The implementation of these metrics is available at [76, 75].

This implementation has been tested on:

- Node Packages
  LCOM2: See graph here EO-LCOM2
  LCOM3: See graph here EO-LCOM3

3.4.3 LCOM4

LCOM4 measures the number of "connected components" in a class. A connected component is a set of related methods (and class-level variables). There should be only one such component in each class. If there are two or more components, the class should be split into so many smaller classes. In some cases, a value that exceeds 1 does not make sense to split the class (e.g., a class implements a form or a web page) as would affect the user interface. The explanation is that they store information in the underlying object that may be not directly using in the class itself. This implementation has been tested on:

- Node Packages
  See graph here EO-LCOM4

3.4.4 LCOM5

'LCOM5' is a 1996 revision by B. Henderson-Sellers, L. L. Constantine, and I. M. Graham [8] of the initial LCOM metric proposed by MIT researchers. The values for LCOM5 are defined in the real interval [0, 1] where '0' means "perfect cohesion" and '1' means "no cohesion". Two problems with the original definition [110] are:

1. LCOM5 produces values across the full range and no specific value has a higher probability than any other (the original LCOM has a preference towards the value "0").

2. Following on from the previous point, the values can be uniquely interpreted in terms of cohesion, suggesting that they are percentages of the "no cohesion" score '1' [8]

The implementation of this metric is available at [77] and has been tested on:

- Node Packages
  see graph here EO-LCOM5

3.4.5 CAMC

In the CAMC metric, the cohesion in the methods of a class is determined by the types of objects (parameter access pattern of methods) these methods take as input parameters. The metric determines the overlap in the object types of the methods' parameter lists. The amount of overlap in object types used by the methods of a class can be used to predict the cohesion of the class. This information is available when all method's prototypes have been defined, well before a class' methods are completely implemented [14].

The CAMC metric measures the extent of intersections of individual method parameter type lists with the parameter type list of all methods in the class. To compute the CAMC metric value, an overall union (T) of all object types in the parameters of the methods of a class is determined. A set M_i of parameter object types for each method is also determined. An intersection (set P_i) of M_i with the union set T is computed for all methods in the class. A ratio 7 of the size of the intersection (P_i) set to the size of the union set (T) is computed for all methods. The sum of all intersection sets P_i is divided by product of the number of methods and the size of the union set T , to give a value for the CAMC metric [14]. For a class with ' n ' methods If M_i is the set of parameters of method i , then T = Union of M_i , i = 1 to n, iff P_i is the intersection of set M_i with T_i i.e.,

$$P = M_i \cap T_i$$

, then

$$CAMC = \frac{1}{kl}\Sigma\Sigma O_{ij} = \frac{\sigma}{kl}$$

The implementation of this metric is available at [67].
This implementation has been tested on:

- Node Packages
  See graph here EO-CAMC

3.4.6 TCC and LCC

Connectivity between methods (CC)

: The direct connectivity between methods is determined by the class abstraction. If there exists one or more common instance variables between two method abstractions then the two corresponding methods are directly connected. Methods that are connected through other directly connected methods are indirectly connected. The indirect connection is the transitive closure of direct connection relationship. Thus, a method M1 is indirectly connected with a method Mn if there is a sequence of methods M_2 , M_3 , . . ., M_{n-1} such that M_1 δ M_2 ; ...; M_{n-1} δ M_n where M_i δ M_j represents a direct connection.

The implementation of these metrics is available at [72, 83].

Parameter of TCC and LCC

let NP(C) be the total number of pairs of methods. NP is the maximum possible number of direct or indirect connections in a class. If there are N methods in a class C:

$$NP(C) = N * (N - 1)/2,$$

let NDC(C) be the number of direct connections, and NIC(C) be the number of indirect connections. **Tight class cohesion (TCC)** is the relative number of directly connected methods:

$$TCC(C) = NDC(C)/NP(C).$$

Loose class cohesion (LCC) is the relative number of directly or indirectly connected methods:

$$LCC(C) = (NDC(C) + NIC(C)) = NP(C).$$

This implementation of TCC has been tested on:

- Node Packages
  TCC: See graph here EO-TCC
  LCC: See graph here EO-LCC

3.4.7 NHD

The hamming distance (HD) metric was introduced by Counsell et al [27]. Informally, it provides a measure of disagreement between rows in a binary matrix. The definition of HD leads naturally to the Normalised Hamming Distance (NHD) metric [27], which measures agreement between rows in a binary matrix. Clearly, this means that the NHD metric is an alternative measure of the cohesion to CAMC metric. The parameter agreement between methods m_i and m_j is the number of places in which the parameter occurrence vectors of the two methods are equal. The parameter agreement matrix A is a lower triangular square matrix of dimension k - 1, where a_{ij} is defined as the parameter agreement between the methods i and j for $1 < j < i < k$, and 0 otherwise.

The implementation of this metric is available at [78] and has been tested on:

- Node Packages
  See graph here EO-NHD

3.4.8 SCOM

The Sensitive Class Cohesion Metrics (SCOM) is a ration of the sum of connection intensities $C_{(i,j)}$ of all pairs (i, j) of m methods to the total number of pairs of methods. Connection intensity must be given more weight $\alpha_{(i,j)}$ when such a pair involves more attributes. SCOM is normalized to produce values in the range [0..1], thus yielding meaningful values [106, 61]:

1. Value zero means no cohesion at all. Thus, every method deals with an independent set of attributes.
2. Value one means full cohesion. Thus, every method uses all the attributes of the class.

$$SCOM = \frac{2}{m(m-1)} \sum_{i=1}^{(m-1)} \sum_{j=i+1}^{m} (C_{i,j}\alpha_{i,j})$$

The implementation of this metric is available at [81].
This implementation has been tested on:

- Node Packages
  See graph here EO-SCOM

3.4.9 CCM

$$CCM = \frac{NC(C)}{NMP(C).NCC(C)},$$

where NC(C) is the number of actual connections among the methods of class, *NMP(C)* is the number of the maximum possible connections among the methods of the class C, *NCC(C)* is the number of connected components of the connection graph G_c that represents each method as a node and two methods A and B are connected in the connection graph if A and B access one or more attributes in common, method A invokes method B or vice versa, or methods A and B invoke one or more methods in common [52]. The implementation of this metric is available at [68] and has been tested on:

- Node Packages
  See graph here EO-CCM

3.4.10 OCC and PCC

If two or more methods have access to one attribute directly or indirectly, those methods seem to share the attribute. Such sharing is a kind of connections among methods. OCC quantifies the maximum extent of such connections within a class. This would be the maximum size of cohesive part of the class [21]. The weak-connection graph represents attribute-sharing. Furthermore, accesses to attributes include data-reading and data-writing. If a method writes data into an attribute, and another method reads data from the attribute, then these methods are dependent. Such relationship is a strong-connection graph. When methods have access to attributes, those accesses include data-readings and data-writings. By focusing on such differences in accesses, we can consider dependent relationships among methods, which would be strong connections among methods. PCC quantifies the maximum extent of such dependent relationships within a class. This would be the maximum size of the highly cohesive part of the class [21].

The implementation of OCC and PCC metrics are available at [79, 80] and PCC been tested on:

- Node Packages
  PCC: See graph here EO-PCC
  OCC: See graph here EO-OCC

.

3.4.11 Comparison of Jpeek-Java vs Jpeek-EO

The comparison of JPeek-EO with JPeek-Java,in terms of runtime performance,is available in this attached file:
Go to `https://github.com/HSE-Eolang/hse-eolang.github.io/blob/main/performance_assesment.xlsx` to download the excel workbook. Overall, the assessments shows that the JPeek-EO runs slower than JPeek-Java.

3.5 Conclusion

The proposed CloudBU module updated certain cohesion metrics originally implemented in JPeek now in EO. The module calculates specific JPeek metrics by the EO programming language and its library. This module was built into the framework of the JPeek tool.

Certain metrics in the original JPeek had different results from that of EO-JPeek. These include LCOM4, CCM, SCOM, TCC, and LCC. During the development of these metrics in the EO programming language, our team ensured correctness by doing some manual checks, which showed that the corresponding EO-JPeek metrics were correct.

Also, while developing the module, a number of bugs in the EO compiler and it's standard library were fixed, which improved the compiler performance. The EO-JPeek application is available at [71]. Testing this module proved that for certain EO use cases, reliability is comparable to that of Java.

Stage 4

Analysis of Eolang efficiency

4.1 Abstract

EO programming language is a new Object-Oriented Programming language that has been going through many phases of development in this research. In this phase, we compare Eolang efficiency to C++ and Java; detect differences; identify pros and cons by SWOT analysis; define metrics to compare, and write test cases as well as automate testing and bench-marking to run for different EO compiler versions. The main goal is to analyze Eolang efficiency.

4.2 Introduction

This report includes metrics analysis, SWOT analysis, benchmark description and results. Benchmarks and tests are available at GitHub repository and integrated into release pipeline in the form of Continuous Integration (CI).

4.3 Comparison Metrics

4.3.1 Criteria for comparison

There are many criteria important to comparing or evaluating general purpose programming languages:

1. **Simplicity of language constructs**.

   The simplicity of a language design[26] includes such measurable aspects as the minimality of required concepts and the integrity and consistency of its structures. Simpilicity here relates to ease of programming. Simple is beautiful is the golden mantra in programming [109]. While efficiency and performance are major factors, simplicity and maintenance cost wins over them in many use cases. These become a deciding factor while choosing a programming language, exploring features in a language or even deciding on standard coding practices within an organization.

   Binary result: yes/no.

2. **Readability**.

   Readability refers to the ease with which codes can be read and understood [39].This relates to maintainability, an important factor as many programs greatly outlive their expected lifetimes.

   Binary result: yes/no.

3. **Compilation speed**.

   This metric identifies the program's total execution time(m.s.ms) [97]. This can comparatively help determine how long it takes to run an algorithm in different languages.

   Result: 1..5, 5-high speed,1-low speed

4. **Memory usage**.

   The memory usage criteria identifies the total amount of memory that were used due to program execution. Some programming languages may be efficient in memory consumption while others are not. This criteria is important for clarifying the differences between EO, C++ and Java.

   Result: 1..5, 5-low usage,1-high usage.

5. **LOC (Lines of code)**.

   This refers to the total number of LOC that were used to develop a program. This metric is used to measure the size and complexity of a software project. It is measured by counting the number of lines in text of the program's source code. LOC can be used to predict the amount of effort that would be required to develop a program, as well as estimate programming productivity or maintainability once the software is developed.

   Result:1..5, 5-high amount,1-low amount.

6. **Debugging help**.

   This criteria helps clarify the amount of tools and help available for the process of detecting and removing of existing and potential errors or bugs in a software code that can cause it to behave unexpectedly or crash.

   Binary result: yes/no.

In order to compare Eolang efficiency to C++ and Java and detect the differences, we select a number of algorithms to implement in all three languages. The following algorithms were chosen:

1. Prim's algorithm
2. Dijkstra's algorithm
3. Kruskal's algorithm
4. Ford-fulkerson's algorithm

The justification for choosing these specific algorithms is that graph algorithms are usually complex, contain loops and recursive calls, and can be tested and compared based on different criteria.

The test results of the implementation of these algorithms are capture under the following headings:

1. vNum - the number of edges
2. Time - the time of program execution
3. Memory - the total memory used while program execution

4.3.2 Prim's algorithm

Prim's algorithm is an algorithm for constructing the minimum spanning tree of a connected weighted undirected graph.

The input of the algorithm is a connected weighted undirected graph, which is represented by a sequence of integers separated by spaces. Each group of 3 numbers describes an arc of the graph: the first two numbers are adjacent vertices, the third is the weight of the edge.

Ex. 0 2 4 2 4 1 4 1 5 1 3 2 3 0 7

First, a random vertex is selected. The edge of the minimum weight incident to the selected vertex is searched for. This is the first edge of the spanning tree. Next, an edge of the graph of the minimum weight is added to the tree in which only one of the vertices belongs to the tree. When the number of tree edges is equal to the number of nodes in the graph minus one (the number of nodes in the graph and the tree are equal), the algorithm ends.

The result of the work of programs implementing the algorithm is a sequence of edges of the minimal spanning tree: Ex. (2 4 - 1) (0 2 - 4) (4 1 - 5) (1 3 - 2)

Time Complexity: O((V + E) log V)

Testing results:

See Fig. 4.1. for test results of [89]

```
========== Prim's algorithm ===========
| C++:                                 |
----------------------------------------
| vNum | Time       | Memory(Kbyte)   |
----------------------------------------
| 10   | 0:00.00    |            3256 |
----------------------------------------
| 30   | 0:00.00    |            3312 |
----------------------------------------
| 50   | 0:00.01    |            3356 |
----------------------------------------
| Java:                                |
----------------------------------------
| vNum | Time       | Memory(Kbyte)   |
----------------------------------------
| 10   | 0:00.05    |           36176 |
----------------------------------------
| 30   | 0:00.12    |           36516 |
----------------------------------------
| 50   | 0:00.09    |           37848 |
----------------------------------------
| EO:                                  |
----------------------------------------
| vNum | Time       | Memory(Kbyte)   |
----------------------------------------
| 10   | 0:00.33    |          121636 |
----------------------------------------
| 30   | 0:03.55    |          601572 |
----------------------------------------
| 50   | 0:20.52    |          812628 |
----------------------------------------
```

Figure 4.1: Prim's algorithm test results

4.3.3 Dijkstra's algorithm

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a SPT (shortest path tree) with a given source as a root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source [57]. Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

1. Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.

2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3. While sptSet doesn't include all vertices

   a) Pick a vertex u which is not there in sptSet and has a minimum distance value.

   b) Include u to sptSet.

   c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u

(from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Time Complexity: O(E log V)

Testing results:

See Fig. 4.2. for the test result of [67]

```
======== Dijkstra's algorithm =========
| C++:                                 |
----------------------------------------
| vNum | Time       | Memory(Kbyte)   |
----------------------------------------
| 10   | 0:00.00    |            3296 |
----------------------------------------
| 30   | 0:00.00    |            3368 |
----------------------------------------
| 50   | 0:00.00    |            3364 |
----------------------------------------
| Java:                                |
----------------------------------------
| vNum | Time       | Memory(Kbyte)   |
----------------------------------------
| 10   | 0:00.09    |           37592 |
----------------------------------------
| 30   | 0:00.09    |           35860 |
----------------------------------------
| 50   | 0:00.13    |           36788 |
----------------------------------------
| EO:                                  |
----------------------------------------
| vNum | Time       | Memory(Kbyte)   |
----------------------------------------
| 10   | 0:00.15    |           51972 |
----------------------------------------
| 30   | 0:00.43    |          112568 |
----------------------------------------
| 50   | 0:00.48    |          127216 |
----------------------------------------
```

Figure 4.2: Dijkstra's algorithm Test results

4.3.4 Kruskal's algorithm

The Kruskal's algorithm is used to find the minimum spanning tree of a graph [55]. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree. A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

To find the MST using the Kruskal's algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. (Union-Find algorithm) Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step no. 2 until there are (V-1) edges in the spanning tree.

Time Complexity: O(E log V)

Testing results:

See Fig. 4.3. for test of results of [87]

```
========= Kruskal's algorithm =========
| C++:                                 |
----------------------------------------
| vNum | Time       | Memory(Kbyte)   |
----------------------------------------
| 10   | 0:00.00    |            3344 |
----------------------------------------
| 30   | 0:00.00    |            3288 |
----------------------------------------
| 50   | 0:00.00    |            3356 |
----------------------------------------
| Java:                                |
----------------------------------------
| vNum | Time       | Memory(Kbyte)   |
----------------------------------------
| 10   | 0:00.09    |           39576 |
----------------------------------------
| 30   | 0:00.19    |           39700 |
----------------------------------------
| 50   | 0:00.18    |           39900 |
----------------------------------------
| EO:                                  |
----------------------------------------
| vNum | Time       | Memory(Kbyte)   |
----------------------------------------
| 10   | 0:00.25    |           67488 |
----------------------------------------
| 30   | 0:01.02    |          266308 |
----------------------------------------
| 50   | 0:03.02    |          809048 |
----------------------------------------
```

Figure 4.3: Kruskal's algorithm test results)

4.3.5 Ford-Falkerson Algorithm

The Ford-Falkerson Algorithm is used to solve the maximum flow problem. This algorithm depends on the concept of residual graph. [58].

To understand and implement it, let us first look at the concept of Residual Graph.

Residual Graph of a flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow. Every edge of a residual graph has a value called residual capacity which is equal to original capacity of the edge minus current flow. Residual capacity is basically the current capacity of the edge. Let us now talk about implementation details. Residual capacity is 0 if there is no edge between two vertices of residual graph. We can initialize the residual graph as original graph as there is no initial flow and initially residual capacity is equal to original capacity. To find an augmenting path, we can either do a BFS or DFS of the residual graph. We have used DFS in below implementation. Using DFS, we can find out if there is a path from source to sink.

The important thing is, we need to update residual capacities in the residual graph. We subtract path flow from all edges along the path and we add path flow along the reverse edges We need to add path flow along reverse edges because may later need to send flow in reverse direction.

Time Complexity:

Time complexity of the above algorithm is O(max_flow E).

Testing results:

See Fig. 4.4. for test of results of [86]

```
====== Ford-Fulkerson algorithm =======
| C++:                                |
---------------------------------------
| vNum | Time       | Memory(Kbyte)   |
---------------------------------------
| 10   | 0:00.00    |            3456 |
---------------------------------------
| 30   | 0:00.01    |            3372 |
---------------------------------------
| 50   | 0:00.00    |            3424 |
---------------------------------------
| Java:                               |
---------------------------------------
| vNum | Time       | Memory(Kbyte)   |
---------------------------------------
| 10   | 0:00.03    |           34280 |
---------------------------------------
| 30   | 0:00.04    |           34424 |
---------------------------------------
| 50   | 0:00.09    |           35740 |
---------------------------------------
| EO:                                 |
---------------------------------------
| vNum | Time       | Memory(Kbyte)   |
---------------------------------------
| 10   | 0:05.92    |         1319204 |
---------------------------------------
| 30   | 0:05.71    |          468860 |
---------------------------------------
| 50   | 1:57.33    |          469332 |
---------------------------------------
```

Figure 4.4: Ford Falkerson's algorithm test results

4.3.6 Comparison Metrics - *Algorithms*

Language	Metric	Grade
C++	Simplicity	yes
Java	Simplicity	yes
EO	Simplicity	no
C++	Readability	yes
Java	Readability	yes
EO	Readability	no
C++	Compilation speed	5(time:3 ms.)
Java	Compilation speed	4(time:1.18 s.)
EO	Compilation speed	1(time:30 s.)
C++	Memory usage	5(memory:30 mb.)
Java	Memory usage	3(memory:340 mb.)
EO	Memory usage	1(memory:2.96 gb.)
C++	LOC	5 (350 LOC)
Java	LOC	3 (500 LOC)
EO	LOC	1 (770 LOC)
C++	Debugging help	yes
Java	Debugging help	yes
EO	Debugging help	yes

4.4 SWOT Analysis

The criteria listed above are equally important because they affect the development cost and effort required over the lifetime of the program, and also affect the usefulness and quality of the developed program. It is important to note that several points are difficult to compare for several reasons, as far as various programming languages are concerned. One of the controversial points is that Eolang is generally positioned as a language intended for static code analysis, which, however, is not explicitly advertised and there is no special emphasis on this in the current project. However, this leads to the fact that initially, it is essentially a subject-oriented and not a universal language that does not allow effectively displaying not only the styles of writing programs but also having a much smaller set of expressive means for describing algorithms and data.

4.4.1 STRENGTHS

Here, in general, a fairly simple semantic model of the language can be noted, which is due to the initially laid down idea of forming an "elegant" programming style. Based on this, the program contains only objects, the semantics of which allows them to be used as actions (directives) that provide both a description of the functionality of algorithms and the structuring of data. This allows you to form a fairly compact **semantic model**, which is the strength of the language.

The solutions proposed in the language increase the reliability of the generated code, albeit often at the expense of efficiency. But for the main target of analysis and reliability improvement, this is not a significant factor.

When compared to C ++ and Java languages, it can be noted that in these languages there are many unreliable constructs for programming. The languages themselves have many redundant and overlapping constructs, which often do not allow the generation of unambiguous and reliable code.

Compactness, extensibility and openness can be used to describe the strength of Eolang.

4.4.2 WEAKNESSES

One of the weaknesses is that the limited capabilities of the language do not allow it to be used in many subject areas in comparison with C ++ and Java. That is, where high performance computing is required.

Another point to mention is **lack of tools** that provide support for parallel and distributed computing, which is currently used in one form or another in almost all modern programming languages.

And not enough attention is paid to the formation of the type system. Using a typeless solution can turn out to be unreliable in many situations, which, in turn, may lead to difficulties associated with static code analysis. In addition, in some cases, to increase control over data when writing programs in Eolang, additional constructions will have to be introduced to model data types and explicitly check them either during static analysis or at runtime.

For C++ and Java, static typing is used to control the data at compile time. In addition, these languages support dynamic typing due to Object-Oriented polymorphism and the possibility of dynamic type checking at runtime.

Currently, the EO community is limited. There are not many conferences, local meetups, forums, Facebook groups, open-source projects based on the language and people willing to help.

Also, as mentioned in section 4.3.6, Eolang lacks **simplicity**. The EO language is barely simple to read and write and has a quite steep learning curve for new programmers. In comparison to the other languages in context, corresponding code in Eolang takes consumes more lines **(LOC)**. The comparison made in section 4.3.6 shows that Eolang has minimal efficiency in terms of **memory consumption** and **compilation speed**. Thus, Eolang consumes a lot of resources, and increasing becomes complex as the code base gets large. These add to the weaknesses of the language.

Lastly, conceptual incompleteness. The Eolang concept or idea is not fully or completely described.

4.4.3 OPPORTUNITIES

Most likely it is prudent to start with the possibilities since they determine the specifics of the language. The limitations through the OO paradigm and recursive computations based on the absence of object mutability possibly simplify the code for static analysis, but at the same time significantly reduces the number of effective techniques used in real programming. When creating algorithms, you often have to write longer and more inefficient code, which is difficult to further optimize when reduced to a real executor. At the same time, as the practice of using functional programming languages shows, the use of similar techniques increases the reliability of programs and ensures the formation of controlled code.

When comparing with C ++ and Java, it is enough to note here that both proposed languages are universal and include tools for writing programs that allow one to choose between reliable and efficient programming. In principle, it is possible to list these tools, emphasizing what is not in Eolang, emphasizing that this significantly expands the possibilities of programming, but often at the expense of the reliability of the code.

Also, the platform independence of Eolang provides an opportunity to potentially interoperate the language with many other languages and use existing libraries.

Additionally, there is potential **formalizability of semantics**, the presence of formal calculus of objects, the convergence of object and functional paradigms, potentially short and reliable code.

4.4.4 THREATS

Among the main threats, we could consider the use of the language not for its main purpose, which can lead to writing code that will be less expressive than the code written in C ++ and Java. At the same time, attempts to model the constructs of these languages in Eolang can lead to more cumbersome and less reliable code. In particular, explicit modeling of data types will require validation at runtime or may lead to the development of additional analysis programs to isolate and match data types during the static analysis phase.

It may also be worth noting the problem associated with the lack of type control when entering data when the incoming data in the presence of a **typeless** language model will be difficult to control. There are similar threats in other languages, but the presence of a static type system or explicit dynamic typing allows the ability to control the input and transformation of data directly using language constructs without additional modeling. Also, the typeless nature of the language drives it more in the direction of functional approach [1] rather than OOP, as intended. It would be threatening to use Eolang as a system programming language since it lacks strong typing, compared to many system programming languages which rather are strongly typed to help manage complexity [13].

4.5 Continuous Integration Testing on GitHub

As one of the results of this stage, we propose a semi-autonomous testing solution integrated into the GitHub Actions. Besides, the testing solution is based on YAML configurations, Python , and JSON. YAML is used to configure GitHub Actions Pipelines. Python is the main language used to implement the semi-autonomous testing framework. JSON is used to configure tests to run them with the proposed testing framework.

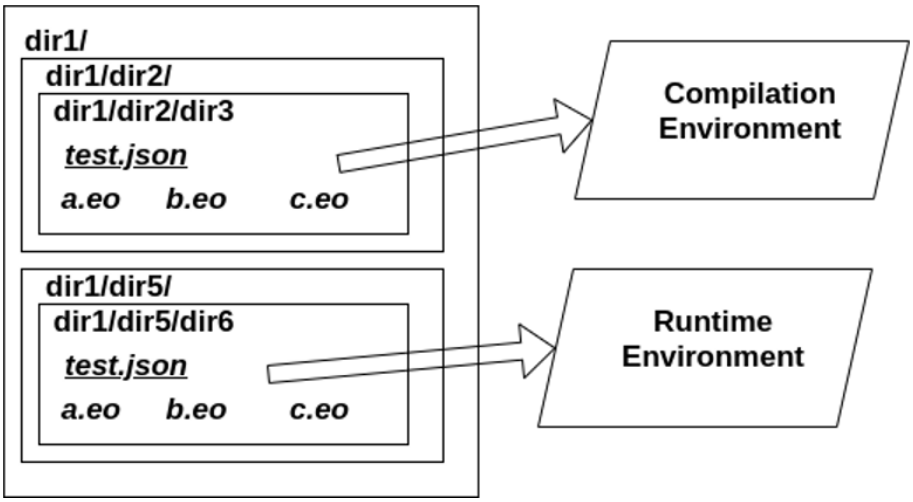

Figure 4.5: CI model

Fig. 4.5. shows the idea behind the proposed testing framework. The framework takes the following steps during its work:

1. GitHub Actions pipelines are triggered and activated. There are two pipelines, one for each tested compilers (CQFN and HSE).
2. The pipeline runs

   python/retrieve-tests-matrix.py to build the tests matrix. Pipeline matrices allow us to dynamically configure the testing grid in GitHub actions. This means that one YAML configuration file can produce an arbitrary number of jobs at runtime. To build the tests matrix, the framework recursively traverses the eo-tests directory and collects test.json files. Each test.json contains the following metadata: a name, a description, a type, an expected result, an activity flag. After the tests metadata is collected, GitHub Actions initialize one job per each test.json.
3. Inside each test's job, GitHub Actions runs python/run-test.py to run the corresponding test.
4. There are three types of tests in total: compilation tests, runtime tests, and compilation time tests. To run tests, four environments are prepared in /environments, one per each test type and compiler (HSE or CQFN). The testing framework copies program files into the corresponding environment and performs testing by running the environment and comparing the expected result to the actual result.

In addition to the test framework, we prepared seventy test cases that check different aspects of the language, including essential operations of abstraction, application, decoration, and dataization. The HSE compiler passed 44 out of 70 tests, while the CQFN compiler passed only 15. The HSE compiler failed at the tests that check language features not yet supported by the compiler (these are partial application, named attribute binding, exclamation mark syntax, decoratee, and parent reference in application syntax). The CQFN tests failed because of incompatibility of the tests with the compiler and the testing system itself, mostly. To conclude, we proposed the semi-autonomous testing solution that allows EO maintainers to add tests written in EO (both compilation-time and run-time tests) in a flexible manner through the incorporation of JSON configuration files. In addition to the framework, we prepared 70 test samples. Testing showed that CQFN and HSE compilers differ in their supported language and runtime features. In the future, we plan to synchronize the codebases of compilers and address issues found through the testing. The proposed framework is delivered as a GitHub repository[64].

4.6 Conclusion

Time and memory efficiency tests show the huge differences between EO and C++/Java. The ratio of Memory usage between EO and C++ is 200. This means that EO programs used 200 times more memory than C++ programs. The ratio of Memory ratios between EO and Java is 20. It is 10 times less than the EO/C++ ratio but still, not efficient. The Time ratio between EO and C++ is 180. This means that EO programs need 180 times more time to execute than C++ programs. The several time ratio between EO and Java is 10. It is 18 times less than the EO/C++ ratio. It can be estimated that the more complex structure or algorithm is implemented in EO, the less Runtime efficiency is. Therefore, the higher the complexity of the algorithm is, the more complex the structure of EO code gets, and the more difficult it becomes to read. Thus, the resulting code loses readability. This further leads to debugging problems, as it becomes more difficult to track and identify bugs or problems in a large amount of EO code. Code is available at [70]

Stage 5

Analysis of Design Patterns

5.1 Abstract

Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system. Eolang programming language is a new Object-Oriented Programming language that has been going through many phases of development in this research. In this phase, we analyze typical design patterns in Java and C++ by researching multiple popular open-source repositories, to detect common design patterns and their usage statistics and then suggest alternatives in Eolang that could replace such patterns which are not supported in the language. We additionally provide explanation why Eolang alternative could be better

5.2 Introduction

This report includes pattern analysis, usage statistics, comparison of design patterns implementation in C++ and Java, implementation of some popular design patterns in Eolang, explanation of why Eolang does not support some design patterns, description of alternatives and explanation of why Eolang alternatives could be better.

5.2.1 Backround

Design is one of the most difficult task in software development [36] and Developers, who have eagerly adopted them over the past years [18], needed to understand not only design patterns [9] but the software systems before they can maintain them, even in cases where documentation and/or design models are missing or of a poor quality. In most cases only the source code as the basic form of documentation is available [24]. Maintenance is a time-consuming activity within software development, and it requires a good understanding of the system in question. The knowledge about design patterns can help developers to understand the underlying architecture faster. Using design patterns is a widely accepted method to improve software development [19]. A design pattern is a general reusable solution to a commonly occurring [53] problem in software design [**noauthor**, 17, 12, 103]. A design pattern isn't a finished design that can be transformed directly into code neither are they static entities, but evolving descriptions of best practices [28]. It is a description or template for how to solve a problem that can be used in many different situations. A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints [42, 15]. Design patterns help to effectively speed up development and engineering processes by providing proven development patterns/paradigms. Quality software design requires considering issues that may not be visible until later in the implementation. Reusing design patterns helps to avoid subtle issues that may be catastrophic and help improve code reliability for programmers and architects familiar with the patterns. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem. They help software engineers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs [102, 10]. In short, the advantages of design patterns include decoupling a request from specific operations (Chain of Responsibility and Command), making a system independent from software and hardware platforms (Abstract Factory and Bridge), independent from algorithmic solutions (Iterator, Strategy, Visitor), or avoid modifying implementations (Adapter, Decorator, Visitor) [29]. Design patterns, overall, helps to thoroughly and designed well implemented frameworks enabling a degree of software reusability that can significantly improve software quality [11, 37].

In this paper, we analyse typical design patterns in Java and C++ and detect common patterns and further look at their usage statistics. There are many design patterns in software development and several of them are common to Java and C++. These design patterns come under three main types.

5.3 Design Patterns

5.3.1 Creational design Patterns

These design patterns are all about class instantiation or object creation. These patterns can be further categorized into Class-creational patterns and object-creational patterns. While class-creation patterns use inheritance effec-

tively in the instantiation process, object-creation patterns use delegation effectively to get the job done. Creational design patterns are the Factory Method, Abstract Factory, Builder, Singleton, Object Pool, and Prototype [22, 7].

Use case of creational design pattern

1. Suppose a developer wants to create a simple DBConnection class to connect to a database and wants to access the database at multiple locations from code, generally what developer will do is create an instance of DBConnection class and use it for doing database operations wherever required. Which results in creating multiple connections from the database as each instance of DBConnection class will have a separate connection to the database. In order to deal with it, we create DBConnection class as a singleton class, so that only one instance of DBConnection is created and a single connection is established. Because we can manage DB Connection via one instance, we can control load balance, unnecessary connections, etc.
2. Suppose you want to create multiple instances of similar kind and want to achieve loose coupling then you can go for Factory pattern. A class implementing factory design pattern works as a bridge between multiple classes. Consider an example of using multiple database servers like SQL Server and Oracle. If you are developing an application using SQL Server database as back end, but in future need to change database to oracle, you will need to modify all your code, so as factory design patterns maintain loose coupling and easy implementation, we should go for the factory design pattern in order to achieving loose coupling and the creation of a similar kind of object.

Factory Method:

Factory Method, also known as virtual constructor, is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created [105] in a way such that it doesn't have tight coupling with the class hierarchy of the library [43]. Factory Method is one of the most used design patterns in Java [99]. It is widely used in C++ code and is very useful when you need to provide a high level of flexibility for your code [104].

Abstract Factory

Abstract Factory patterns work around a super-factory which creates other factories. Thus, it defines a new Abstract Product Factory for each family of products. This factory is also called as factory of factories. It provides one of the best ways to create an object. Abstract Factory design pattern covers the instantiation of the concrete classes behind two kinds of interfaces, where the first interface is responsible for creating a family of related and dependent products, and the second interface is responsible for creating concrete products. The client is using only the declared interfaces and is not aware which concrete families and products are created [35]. Adding a new family of products affects any existing class that depends on it, and requires complex changes in the existing Abstract Factory code, as well as changes in the application or client code [35]. Bulajic and Jovanovic [35] demonstrates a solution where adding a new product class does not require complex changes in existing code, and the number of product classes is reduced to one product class per family of related or dependent products.

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern [98]. Abstract Factory pattern is almost similar to Factory Pattern is considered as another layer of abstraction over factory pattern [49].

Abstract factory pattern implementation provides us a framework that allows us to create objects that follow a general pattern. So, at runtime, abstract factory is coupled with any desired concrete factory which can create objects of desired type, [93]. Fig. 5.1 shows a UML class diagram example for an Abstract Factory Design pattern.

This pattern is particularly useful when the client doesn't know exactly what type to create. The Abstract Factory pattern helps you control the classes of objects that an application creates by isolating concrete classes. The class of a concrete factory appears only once in an application, that is where it's instantiated. This makes it easy to change the concrete factory an application uses. Abstract Factory makes this easy for an application use object from only one family at a time when product objects in a family are designed to work together. Abstract Factory interface fixes the set of products that can be created. This serves as a disadvantage because extending abstract factories to produce new kinds of Products is not easy. The abstract factory design pattern can be implemented in both Java and C++ as demonstrated at [49, 92]. The Abstract Factory pattern is pretty common in C++ code. Many frameworks and libraries use it to provide a way to extend and customize their standard components

Builder

Builder pattern builds a complex object using simple objects and using a step-by-step approach and the final step will return the object. The builder is independent of other objects. Fig. 2 show a UML diagram of builder design pattern. Immutable objects can be build without much complex logic in object building process. Builder design pattern also helps in minimizing the number of parameters in constructor and thus there is no need to pass in null for optional parameters to the constructor. As a disadvantage, it requires creating a separate ConcreteBuilder for each different type of Product [101, 51].

Singleton

In software engineering, the term singleton implies a class which can only be instantiated once, and a global point of access to that instance is provided [93]. This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. It one of the simplest design patterns in Java and C++.

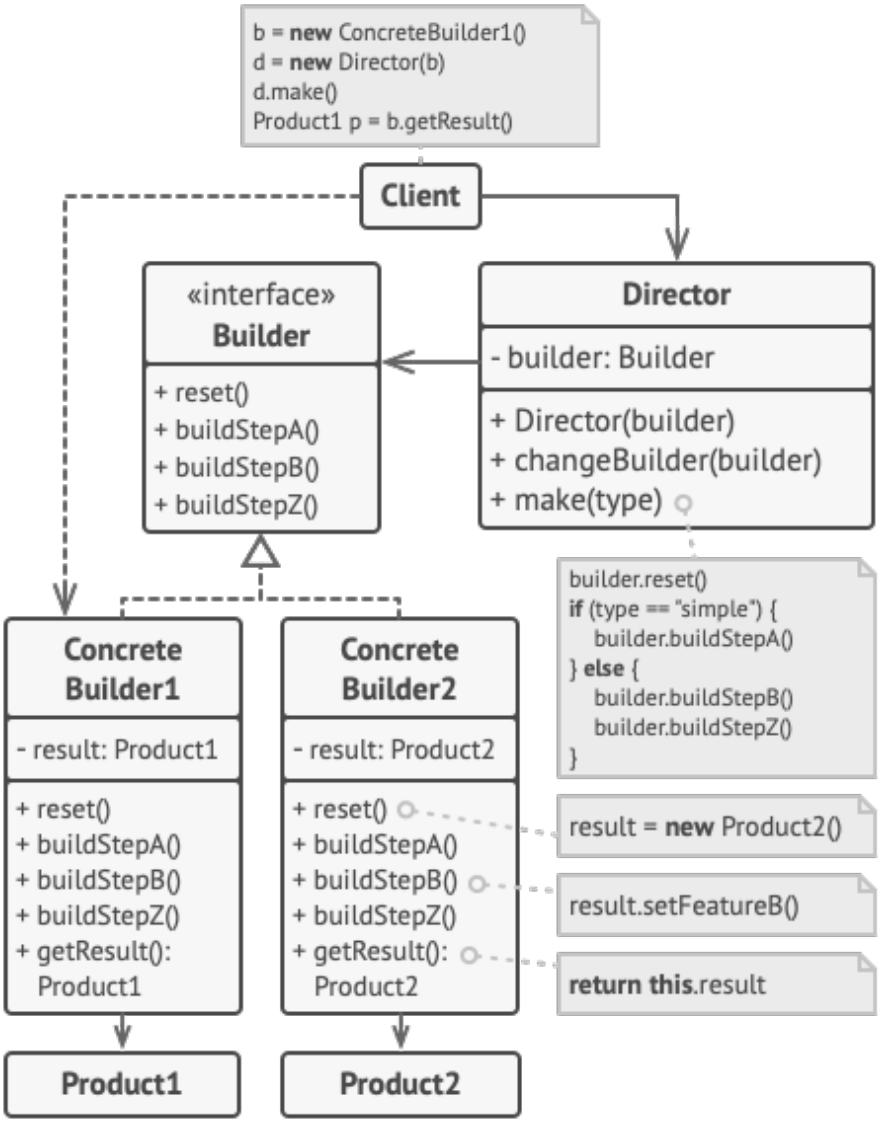

Figure 5.1: UML class diagram example for the Abstract Factory Design Pattern

Object Pool

Object pool pattern is a software creational design pattern which is used in situations where the cost of initializing a class instance is very high. An Object pool is a container which contains some number of objects. So, when an object is taken from the pool, it is not available in the pool until it is put back.

Prototype

Prototype pattern refers to creating duplicate object while keeping performance in mind. This pattern involves implementing a prototype interface which tells to create a clone of the current object.

5.3.2 Structural

These design patterns are about organizing different classes and objects to form larger structures and provide new functionality. Structural design patterns are Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data, and Proxy [22, 7].

Use Case of Structural Design Pattern

When 2 interfaces are not compatible with each other and want to establish a relationship between them through an adapter it's called an adapter design pattern. Adapter pattern converts the interface of a class into another interface or class that the client expects, i.e adapter lets classes works together that could not otherwise because of incompatibility. So, in these types of incompatible scenarios, we can go for the adapter pattern.

Adapter

Adapter design pattern allows objects with incompatible interfaces to collaborate [91]. Adapter pattern works as a bridge between those two incompatible interfaces. A real-life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop [100].

Bridge

The bridge pattern is used when we need to decouple [18] an abstraction from its implementation so that the two can vary independently [91]. It helps you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

Composite

Composite design pattern helps in composing objects into tree structures and then work with these structures as if they were individual objects. It is used where a group of objects need to be treated in similar way as a single object. This pattern creates a class that contains group of its own objects and provides ways to modify this group of same objects.

5.3.3 Decorator

Decorator design pattern allows a user to add new functionality to an existing object without altering its structure. It allows the attachment of new behaviours to objects by placing these objects inside special wrapper objects that contain the behaviours.

Facade

Facade design pattern, as the name goes, hides the complexities of the system and provides an interface to the client by which the client can access the system. It provides a simplified interface to a library, a framework, or any other complex set of classes. In essence, it provides methods required by the client and delegates calls to methods of existing system classes.

Flyweight

The Flyweight pattern enables you to fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all the data in each object. It is primarily used to reduce the number of objects created and to decrease memory footprint and increase performance.

Private Class Data

Private Class Data is used to encapsulate class data and control write access to class attributes as it separates data from methods that us it.

Proxy

The Proxy pattern controls access to the original object [18], allowing you to perform something either before or after the request gets through to the original object. It represents functionality of another class.

5.3.4 Behavioural

Behavioral patterns are about identifying common communication patterns between objects and realizing these patterns. Behavioral patterns are Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method and Visitor [22, 7].

Use Case of Behavioral Design Pattern

The template pattern defines the skeleton of an algorithm in an operation deferring some steps to sub-classes. The template method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure. For example, in your project, you want the behavior of the module to be able to extend, such that we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications. However, no one is allowed to make source code changes to it, i.e. you can add but can't modify the structure in those scenarios a developer can approach template design pattern [102, 42].

Sample implementation of these patterns [107] are available here in both Java and C++.

Chain of responsibility

Chain of responsibility pattern suggests a chain of receiver objects for a request. It decouples sender and receiver of a request based on type of request. It allows you to pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Command

Command pattern is also known as action or transaction pattern. This pattern turns a request into a stand-alone object that contains all information about the request. The transformation allows you pass requests as a method argument, delay or queue a request's execution, and support undoable operations. It is data driven and it wraps an object under an object as command [18] and passes it to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

Interpreter

Interpreter pattern provides a way to evaluate language grammar or expression. Given a language, interpreter defines a representation for the language's grammar along with an interpreter that uses the representation to interpret sentences in the language. Then it maps a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design. This pattern is used in SQL parsing, symbol processing engine etc. The Iterator pattern is very commonly used design pattern in Java and .Net programming environment. It allows sequential traversal through a complex data structure without exposing its internal details [91]. This pattern is also common in C++ code. Mediator is used to reduce communication complexity between multiple objects or classes by providing a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling [91]. It encapsulates how a set of objects interact.

Memento

Memento pattern is used to restore state of an object to a previous state. Without violating encapsulation, you can capture and externalize an object's internal state so that the object can be returned to this state later.

Null Object

In Null Object pattern, a null object replaces check of NULL object instance. The intent of a Null Object is to encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do-nothing behaviour in case data is not available.

Observer

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically [18].

State

In State pattern, objects are created to represent various states and a context object whose behaviour varies as its state object changes. So, in State pattern, a class behaviour changes based on changes in its internal state.

Strategy

Strategy design pattern is the pattern where a class behaviour or its algorithm can be changed at run time.

Template

In Template pattern, an abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need, but the invocation is to be in the same way as defined by an abstract class. It allows the algorithm to vary independently from the clients that use it.

Visitor

In Visitor pattern, a visitor class changes the executing algorithm of an element class. By this way, execution algorithm of element can vary as and when visitor varies. Visitor allows the definition of a new operation without changing the classes of the elements on which it operates.

5.4 Comparison of Design Patterns in C++ and Java

See Table 5.1.

Table 5.1: Comparison of Design Patterns in C++ and Java

Category	Pattern	Java	C++
Creational	Factory method	Uses abstract keyword to declare factory methods in factory classes to be later implemented by subclasses	Uses static pointers to declare factory methods
	Abstract factory	Uses 'abstract' keyword to make abstract factories classes and interfaces	Uses 'class' keyword and pointers to create abstract factories and the 'new' keyword to create concrete factories which is later used to create concrete objects
	Builder	Defines classes with creation methods for creating or building complex objects.	An abstract base class declares the standard construction process, and concrete derived classes define the appropriate implementation for each step of the process. Uses struct and class keyword in process.
	Singleton	Uses the public keyword to define a single point of access method for classes	Make the class responsible for its own global pointer and "initialization on first use" (by using a private static pointer and a public static accessor method)
	Prototype	Uses the cloneable interface for implementing class prototypes	A superclass defines a clone method and subclasses implement this method to return an instance of the class
Structural	Adapter	The adapter class uses the extend keyword to extend another class to make it compatible with another class	An abstract base class is created that specifies the desired interface. An "adapter" class is defined that publicly inherits the interface of the abstract class, and privately inherits the implementation of the legacy component. This adapter class "maps" or "impedance matches" the new interface to the old implementation.
	Bridge	Use abstraction and implementation to decouple classes into several related hierarchies	Uses abstraction and implementation
	Composite	Uses inheritance to hierarchically implement an object tree	Uses inheritance and polymorphism to implement scalar/primitive classes and vector/container classes
	Decorator	Uses interface keyword to wrap subclasses and allow the subclasses to dynamically add new behaviours to objects	Uses the concept of wrapping-delegation which involves pointers to help add new behaviours to objects dynamically
Behavioural	Chain of responsibility	Defines an abstract class with series of methods including abstract methods for other classes to implement and handle a chain of actions independently	Uses pointers and classes and defines a chain method in the base class for delegating to the next object.
	Command	Applies the concept of inheritance and encapsulation to turn actions into objects	Applies the concept of inheritance and encapsulation to turn actions into objects
	Observer	Defines event listeners based on inheritance and encapsulation	Models the "independent" functionality with a "subject" abstraction and Models the "dependent" functionality with "observer" hierarchy
	Null object	Encapsulates the absence of an object by providing a substitutable alternative that offers suitable default do nothing behaviour	Similarly, provides a class that checks for null and returns a boolean
	Mediator	Uses 'interface' keyword to declare mediators which are later implemented by classes that intend to communicate with each other	Uses classes and pointers to implement mediators

5.5 Criticism

The concept of design patterns has been criticized by some in the field of computer science.

5.5.1 Targets the wrong problem

The need for patterns results from using computer languages or techniques with insufficient abstraction ability. Under ideal factoring, a concept should not be copied, but merely referenced. But if something is referenced instead of copied, then there is no "pattern" to label and catalog. Paul Graham writes in the essay Revenge of the Nerds [102].

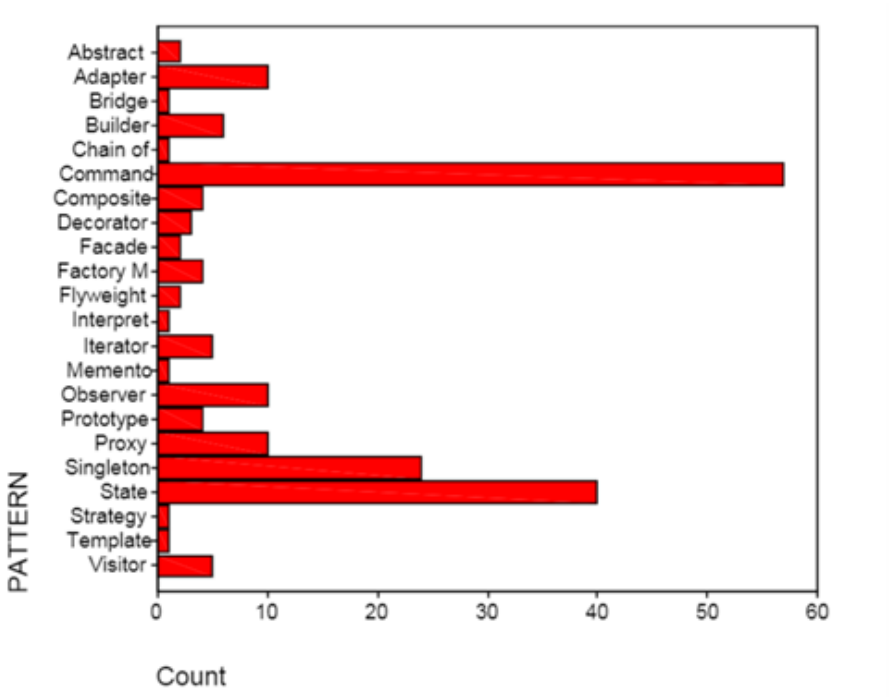

Figure 5.2: Number of Java Projects using individual design patterns [19]

Peter Norvig provides a similar argument. He demonstrates that 16 out of the 23 patterns in the Design Patterns book (which is primarily focused on C++) are simplified or eliminated (via direct language support) in Lisp or Dylan.

5.5.2 Lacks formal foundations

The study of design patterns has been excessively ad hoc, and some have argued that the concept sorely needs to be put on a more formal footing. At OOPSLA 1999, the Gang of Four were (with their full cooperation) subjected to a show trial, in which they were "charged" with numerous crimes against computer science. They were "convicted" by $\frac{2}{3}$ of the "jurors" who attended the trial.

5.5.3 Leads to inefficient solutions

The idea of a design pattern is an attempt to standardize what are already accepted best practices. In principle this might appear to be beneficial, but in practice it often results in the unnecessary duplication of code. It is almost always a more efficient solution to use a well-factored implementation rather than a "just barely good enough" design pattern [102].

5.5.4 Does not differ significantly from other abstractions

Some authors allege that design patterns don't differ significantly from other forms of abstraction, and that the use of new terminology (borrowed from the architecture community) to describe existing phenomena in the field of programming is unnecessary. The Model-View-Controller paradigm is touted as an example of a "pattern" which predates the concept of "design patterns" by several years. It is further argued by some that the primary contribution of the Design Patterns community (and the Gang of Four book) was the use of Alexander's pattern language as a form of documentation; a practice which is often ignored in the literature [102].

According to [36], generality, precision, and understandability are the most important goals to consider in order to simplify software design pattern description.

5.6 Common design patterns in popular open-source repositories and their usage statistics

Hahsler [19] analyses the application of design patterns in Java by identifying patterns in projects using their log messages to look for names and descriptions. This attempt was done based on the idea that the names of design patterns become part of a common design language which developers use to communicate more efficiently. Fig. 5.2 shows the graph of the usage statistics according the approach of Hahsler [19].

Vokac [23] analysed the weekly evolution and maintenance of a large commercial product (C++, 500,000 LOC) over three years, comparing defect rates for classes that participated in selected design patterns to the code at large. He extracted design pattern information and concluded that Observer and Singleton patterns are correlated with

Patterns	Occurrences	%
No Pattern	183634	77.5%
Factory	20237	8.5%
Singleton	3331	1.4%
Observer	16061	6.8%
Template Method	5381	2.3%
Decorator	1513	0.6%
Factory + Observer + Singleton	485	0.2%
Observer + Template	953	0.4 %
Observer + Singleton	2390	1.0%
Factory + Observer	612	0.3%
Factory + Singleton	2279	1.0

Table 5.2: Frequencies of Pattern Occurrences and Percentage of Code Covered by the Patterns

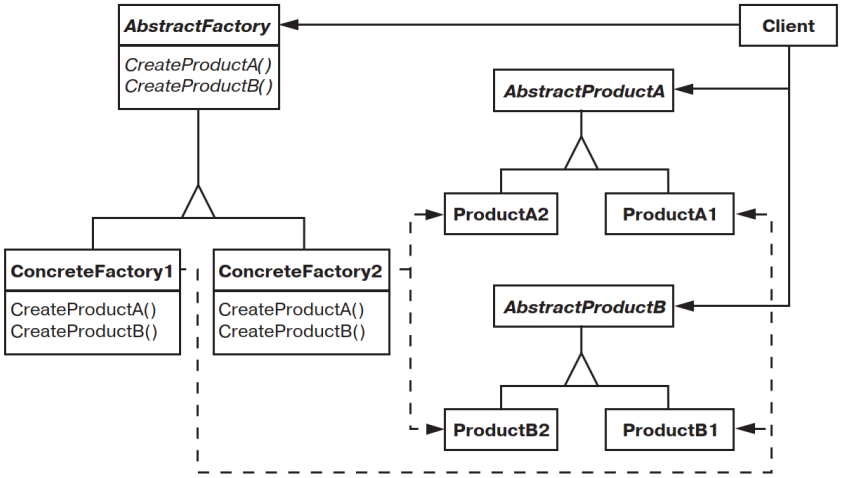

Figure 5.3: Abstract Factory

larger code structures. The Template Method pattern was used in both simple and complex situations, leading to no clear tendency. The frequencies of pattern occurrence are shown in table 5.2.

5.7 Implementation of Some Design Patterns in Eolang

5.7.1 Abstract Factory

An abstract factory is a pattern that generates objects.

Purpose

Provides an interface for creating families of interconnected or interdependent objects without specifying their specific classes.

Structure

See Fig 5.3.

Participants

1. AbstractFactory — abstract factory: declares an interface for operations that create abstract product objects.
2. ConcreteFactory — specific factory: implements operations that create specific objects-products.
3. AbstractProduct — abstract product: declares the interface for the type of object-product.
4. ConcreteProduct — specific product: defines the product object created by the corresponding particular factory, and implements the AbstractProductinterface.
5. Client: uses only interfaces that are declared in the AbstractFactory and AbstractProductclasses.

Implementation

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

[type] > abstractFactory
  if. > concreteFactory
    eq.
      type
      "1"
    concreteFactory1
    concreteFactory2

  [] > createProductA
    createProductA. > @
      ^.concreteFactory
  [] > createProductB
    createProductB. > @
      ^.concreteFactory

[] > concreteFactory1
  [] > createProductA
    1 > @
  [] > createProductB
    2 > @

[] > concreteFactory2
  [] > createProductA
    "one" > @
  [] > createProductB
    "two" > @

[args...] > appAbstractFactory
  abstractFactory > objFactory
    args.get 0
  stdout > @
    sprintf
      "ProductA: %s\nProductB: %s\n"
      objFactory.createProductA
      objFactory.createProductB
```

Output

```
$ ./run.sh 1
ProductA: 1
ProductB: 2
$ ./run.sh 2
ProductA: one
ProductB: two
```

This program creates objects of integers or strings depending on the args parameter [0]. If args[0] = 1, then objects 1 and 2 will be created, otherwise - ”one” and ”two”. The template assumes the use of interfaces that are not present in the EO. In this case, an attempt was made to implement the interface through the EO object has a type parameter depending on which a specific implementation of the object factory is selected. This makes the interface object dependent on the set of implementations of this interface (when adding anew implementation, you must make changes to the interface object).

5.7.2 Singleton (singles)

A singleton is a pattern that generates objects.

Purpose

Ensures that the class has only one instance and provides a global access point to it.

Structure

See fig. 5.4.

Participants

Singleton Singleton:

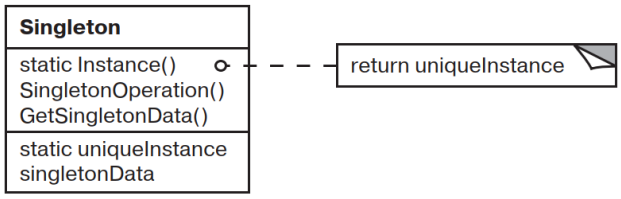

Figure 5.4: Singleton Design Pattern

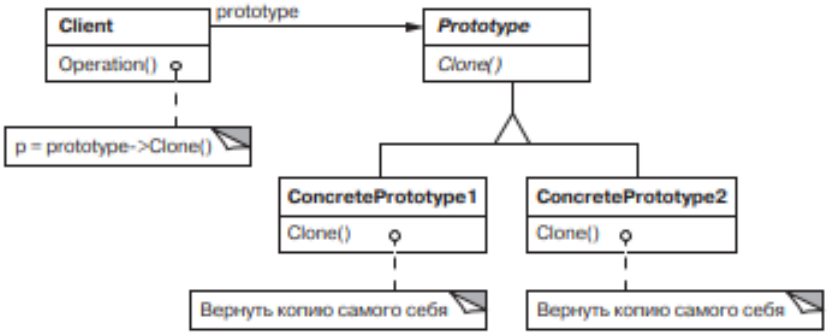

Figure 5.5: Prototype Design Pattern

1. Defines an Instance operation that allows clients to access a single instance. Instance is a class operation, that is, a static method of a class
2. May be responsible for creating your own unique instance.

Relations

Clients access an instance of the Singleton class only through its Instance operation.

Implementation

There are no classes in the EO, so this pattern is not implemented in its pure form. If we define Singleton in terms of EO as an object that is guaranteed to have only one copy, then the implementation of this object is also impossible for the following reasons:

1. There are no references in the EO. Any use of an object in a location other than the place of definition is a copy of this object.
2. EO does not have the ability to restrict access to objects or prevent it from being copied. You cannot restrict the creation of copies of an object.

5.7.3 Prototype

A prototype is a pattern that generates objects.

Purpose

Specifies the types of objects to create using the prototype instance and creates new objects by copying the prototype.

Structure

See fig 5.5.

1. — prototype: declares an interface for cloning itself.
2. — Concrete Prototype: implements the operation of cloning itself.
3. — client: creates a new object by asking the prototype to clone itself.

Implementation

In Eolang, each object can be copied, and each object can perform template functions.

5.7.4 Observer

In EO, all objects have immutable state. Based on the purpose of the template, its use in EO is pointless.

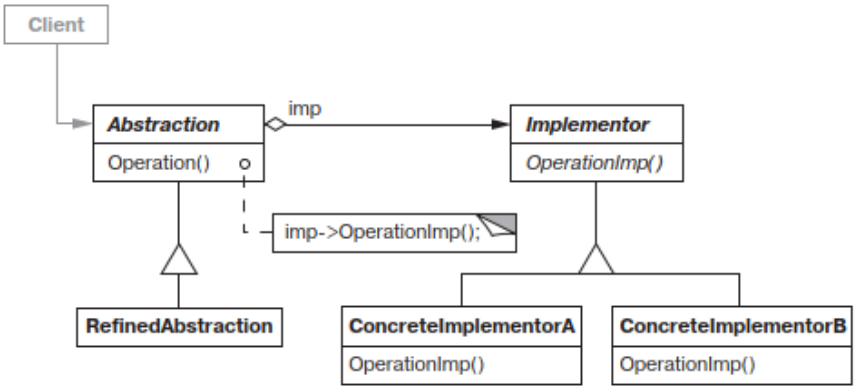

Figure 5.6: Bridge Design Pattern

5.7.5 Bridge

A bridge is a pattern that structures objects.

Purpose

Separate abstraction from its implementation so that both can be changed independently.

Structure

See fig. 5.6

Participants

1. Abstraction — abstraction: defines the abstraction interface, and stores a reference to an object of type Implementor.
2. RefinedAbstraction — refined abstraction: extends the interface defined by abstraction abstraction.
3. Implementor — implementer: defines the interface for the implementation classes. it does not have to exactly match the interface of the abstraction class. In fact both interfaces can be completely different. usually the interface of the Implementor class provides only primitive operations, and the Abstraction class defines higher-level operations based on these primitives.
4. ConcreteImplementor — specific implementer: implements the interface of the Implementor class and defines its specific implementation.

Relations

The Abstraction object redirects client requests to its Implementor object.

5.7.6 Chain of responsibility

A chain of responsibilities is a pattern of behavior of objects.

Purpose

Avoids binding the sender of the request to its recipient by providing the ability to process the request to multiple objects. Binds the receiving objects to the chain and passes the request along that chain until it is processed.

Structure

See fig. 5.7.

Participants

1. Handler — handler: defines the interface for processing requests; (optionally) implements communication with the successor.
2. ConcreteHandler — specific handler: processes the request for which it is responsible; Has access to his successor; If ConcreteHandler is able to process the request, it does so, if it cannot, it sends it to its successor;
3. Client: Sends a request to some ConcreteHandler object in the chain.

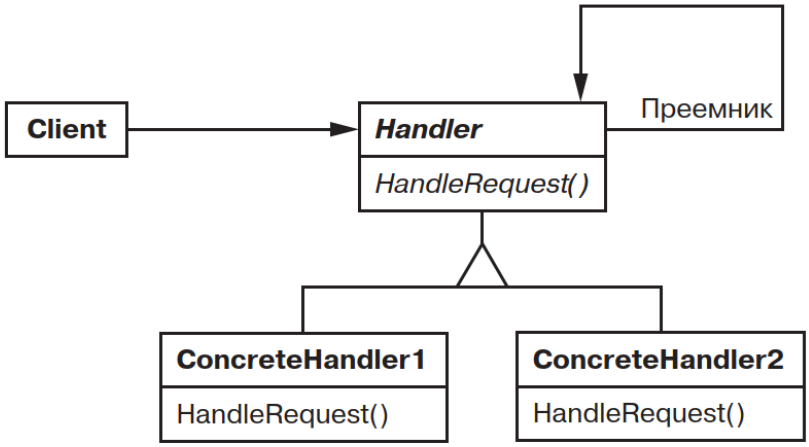

Figure 5.7: Chain of responsibility Design Pattern

Relation

A request initiated by a client is moved along the chain until some ConcreteHandler object takes responsibility for processing it.

Implementation

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

[nextHandler] > defaultHandler
  [message] > process
    "" > @

[] > handler1
  [message] > process
    if. > @
      message.eq "1"
      "one"
      ^.nextHandler.process message
  defaultHandler > @
    handler2

[] > handler2
  [message] > process
    if. > @
      message.eq "2"
      "two"
      ^.nextHandler.process message
  defaultHandler > @
    handler3

[] > handler3
  [message] > process
    if. > @
      message.eq "3"
      "three"
      ^.nextHandler.process message
  defaultHandler > @
    handler4

[] > handler4
  [message] > process
    if. > @
      message.eq "4"
      "four"
      ^.nextHandler.process message
  defaultHandler > @
    defaultHandler
```

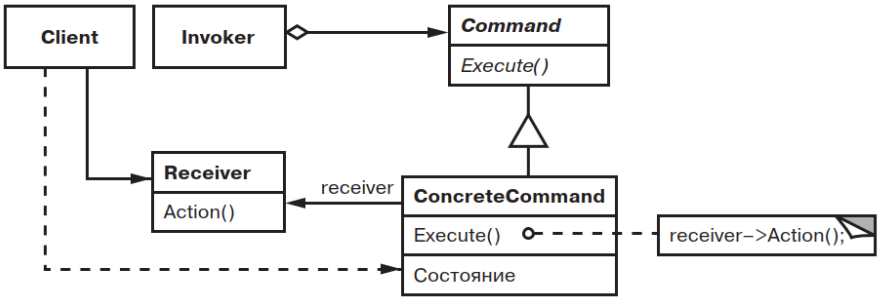

Figure 5.8: Command Design Pattern

```
44.
45. [args...] > appChain
46.   handler1 > hChain
47.   stdout > @
48.     sprintf
49.       "%s\n"
50. hChain. process
51.         args.get 0
```

The input parameter args[0] is passed sequentially to 4 handlers, each of which processes its value (numbers from 1 to 4 are converted into words if another parameter is entered, an empty string is returned).

5.7.7 Command

Command pattern is a behavioral design pattern.

Purpose

Encapsulates a query in an object, thereby allowing clients to be parameterized for different requests, queued or logged requests, and supports cancellation of operations.

Structure

See fig. 5.8

Participants

1. - Command — command: declares the interface to perform the operation.
2. - ConcreteCommand is a specific team: defines the relationship between the Receiver receiving object and the action; implements the Execute operation by calling the corresponding operations of the Receiverobject.
3. — Client— client: Creates a ConcreteCommand class object and sets its recipient.
4. - Invoker— initiator: calls the command to execute the request.
5. - Receiver — recipient: has information about how to perform the operations necessary to meet the request. Any class can act as a recipient.

Relations

1. - The client creates a ConcreteCommand object and sets a recipient for it.
2. - The Invoker initiator saves the ConcreteCommandobject.
3. - The initiator sends a request by calling the ExecuteCommandOperation. If undoing of actions performed is supported, ConcreteCommand stores sufficient status information to perform the cancellation before calling Execute.
4. - The ConcreteCommand object invokes the recipient's operations to execute the request.

5.7.8 Null

The Null Object Pattern is one of the behavioral design patterns.

Purpose

Null object pattern is used to replace check of NULL object instance to simplify the use of dependencies that can be undefined.

Problem

In Null Object pattern, a null object replaces check of NULL object instance. Instead of putting if check for a null value, Null Object reflects a do-nothing relationship. Such Null object can also be used to provide default behaviour in case data is not available. The concept of null objects comes from the idea that some methods return null instead of real objects and may lead to having many checks for null in your code.

In Java and C++, the key to the Null Object pattern is an abstract class that defines the interface for all objects of this type. The Null Object is implemented as a subclass of this abstract class. Because it conforms to the abstract class' interface, it can be used any place this type of object is needed. Null object should not have any state.

In Eolang, the concept of null does not exist as every object is meant to dataize to a value, and as such given a value on creation. As classes and interfaces do not exist here either, the closest implementation in Eolang would be to have every object implement a null attribute that either dataizes to a Boolean or a string representing a lack of value/data or whatever the default value of an object may be. In this case, there may still be checks to see if null is true or false or contains the expected string.

Implementation

```
1. +package sandbox
2. +alias sprintf org.eolang.io.sprintf
3.
4. [] > null
5.   "null" > @
6.
7. [args...] > appNull
8.   sprintf > @
9.     if.
10.       args.isEmpty
11.       null
12.       args.get 0
```

output:
run.cmd null

5.7.9 Decorator

Decorator is a structural design pattern.

Purpose

Decorator pattern allows you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Problem

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

In Eolang, a copy of an object can be made, and new functionality be added. Here, the original object represents the decorator.

Implementation

```
1. +package sandbox
2. +alias sprintf org.eolang.io.sprintf
3.
4. [] > carsDecorate
5.   8 > @
6.
7. [num] > someCars
8.    decorateWithMoreCars num > @
9.
10.    [number] > decorateWithMoreCars
11.      add. > @
12.        carsDecorate
13.        number
14.
15. [args...] > appDecorator
16.   stdout > @
17.     someCars (args.get 0)
```

In this example, the object “someCars” increases the number of cars in decarDecorator is for itself.
Output
run.cmd 5 13
It can be concluded that decorator design pattern naturally exits in Eolang.

5.7.10 Builder

Suppose, we have a class with a variety of input parameters. The input parameters are used to configure instances of the class. Some of the parameters may be optional, while some of them are mandatory to be sat up. Hence, the following techniques of configuration of instances of the class may be applied:

1. Configuration of instance variables of an object directly in the user code through Setter Methods calls or by referencing variables straightforwardly. This practice may not be considered appropriate because it makes code instances more cohesive and interdependent while violating encapsulation of the inner state of objects (which may lead to breaking of the integrity of business logic of an application), and, hence, the usage of the practice is not encouraged. In addition, this technique may allow situations in which objects are being in an incomplete state, which also may break the logic of an application.
2. Creation of subclasses of the considered class when each successor has a slightly changed prototype of its constructors. This technique implies that prototypes of constructors of different subclasses have subsets of optional parameters in them while omitting some parameters, which makes it possible to create configurable instances of objects in a controlled manner. This practice is more encouraged to be applied in practice since it implies control over the creational process. However, it is not recommended for use when choosing the sole parent superclass is challenging or when the practice produces a wide or deep hierarchy of inheritance
3. Overloading of constructors or setting a single constructor with optional parameters. While this practice allows classes to create instances in a controlled way, it is undesirable in cases where the number of parameters or constructor overloads is too large to be manageable and understandable.

The Builder pattern may be considered a universal solution to the problem. The pattern defines the Builder class, which has methods (stages) for building objects. The user code can call the stages in any order, omitting some of them (optional configurations). Also, the Builder class may check that all the required parameters are set up. At the end of construction, user code is required to call a method that finishes the construction process and returns a ready-made object. The pattern encapsulates the creational sensitive logic inside the Builder class and makes the configuration process manageable to the user code.

Structure

See fig. 5.9.

Code Instances Involved

Builder is an abstract class that defines the contract of the creational steps of Products for its successors (concrete builders). Also, the Builder superclass defines the finizaliotion method. Product is an interface for products (instances being created and configured through the Builder pattern). The interface defines the contract for all products so that these may be managed by Builders. (optional) Director is a class, which defines higher-level (that is, higher than the level of ”understanding” of the builder itself, for example, rules for mandatory fields and compliance with business logic) scripts for building objects. The director can be used to reuse some high-level business logic for constructing objects based on various builder implementations.

Relations

Implementations of the Product interface are products. Inheritors of the Builder class provide concrete implementations for the building steps (or borrow some of those steps from the superclass). The Director (optional entity) class can manage builders in a general style (based on some configuration) in accordance with the higher-level logic of business rules. The client code can contact the Director, giving it the configuration, or build an object using the Builder directly.

Implementation

First, we should mention that the problem solved by the Builder pattern may be addressed by the partial application mechanism embedded into the language as one of its features. The partial application mechanism allows programmers to partially apply objects (i.e., create objects with some or all of the input attributes omitted and then, optionally, set unbound attributes after throughout the program). This technique may be utilized as a more concise alternative to constructor overloading. Here is an example:

```
[a b c name] > triangle
  add. > perimeter
    add.
      a
      b
    c

  sprintf > toString
    "The triangle is named '%s'."
    name
```

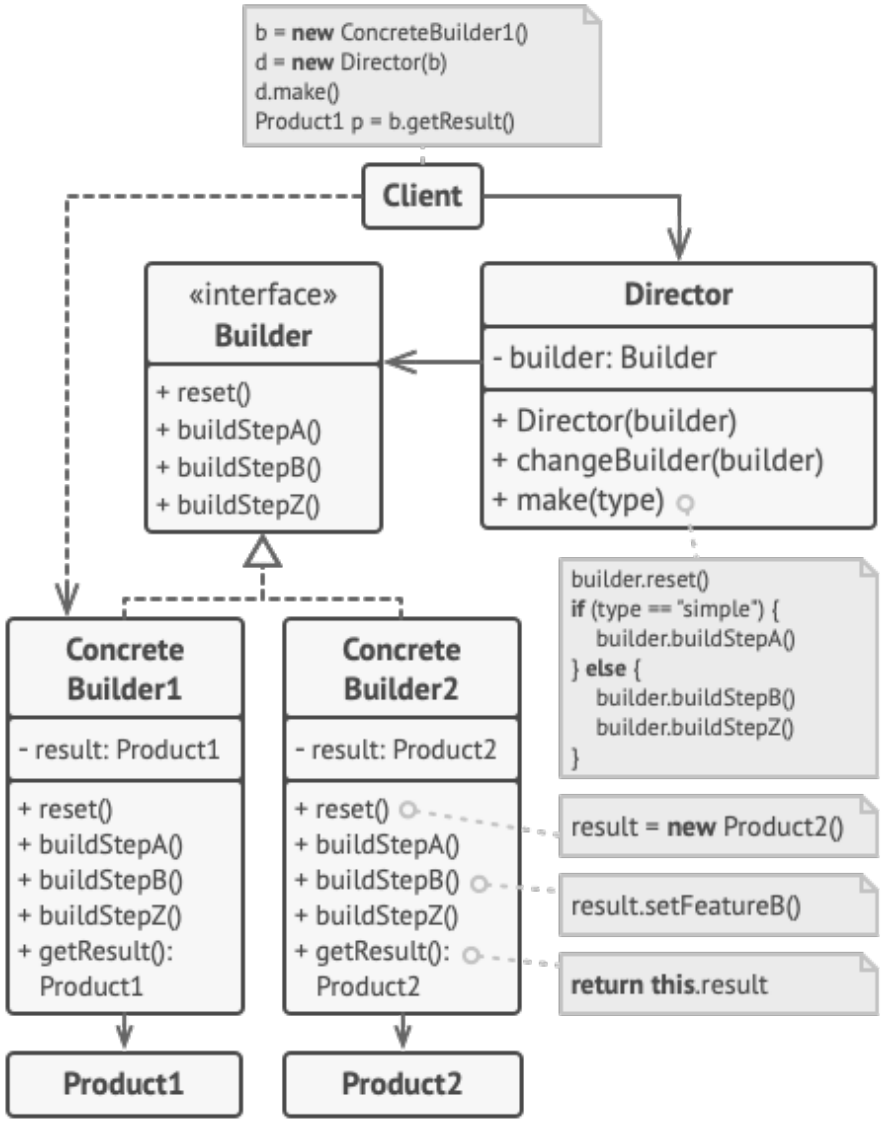

Figure 5.9: Builder Design Pattern

```
[args...] > app
  triangle > triangleA
    10:a
  triangleA > triangleABC
    7:b
    8:c
  triangleABC > triangleABCNamed
   "My Triangle":name
  triangle > triangleNamed
    "My Another Triangle":name
```

Here, we have the triangle object with input attributes a, b, c, and name. The triangle object has two bound attributes: perimeter (which relies on a, b, and c) and toString (which relies on name). Object app demonstrates the partial application mechanism. So, triangleA has only the a attribute bound, triangleABC has all the sides (a, b, c) sat up, triangleABCNamed has all the sides and its name configured, and triangleNamed has the name only. All three triangles are constructed through partial application (meaning, some of the attributes are left unbound or were bound after). The above example demonstrates an alternative solution to the problem of optional configuration of objects. However, this solution does not encapsulate the creation process of objects. Hence, the Builder pattern still may have its place in the EO environment.

Consider the following example:

```
[] > builder
```

```
  subbuilder triangle > @
  [triangleEntity] > subbuilder
    # finalizes the construction process
    [] > finalize
      ^.^.subbuilder > @
    # configures the a free attribute
    [aVal] > setA
      ^.^.subbuilder > @
        ^.^.triangleEntity
          (^.validateSide aVal):a
    # configures the b free attribute
    [bVal] > setb
      ^.^.subbuilder > @
        ^.^.triangleEntity
          (^.validateSide bVal):b
    # configures the c free attribute
    [cVal] > setc
      ^.^.subbuilder > @
        ^.^.triangleEntity
          (^.validateSide cVal):c
    # configures the name free attribute
    [nameVal] > setname
      ^.^.subbuilder > @
        ^.^.triangleEntity
          (^.validateName nameVal):name
    # validates side value
    [val] > validateSide
      if. > @
        val.greater 0
        val
        error
          "The side of a triangle must not be less than 1!"
    # validates name
    [val] > validateName
      if. > @
        val.length.neq 0
        val
        error
          "The name of a triangle must not be empty!"

[args...] > app
  builder > b
  finalize. > triangleABC
    setC.
      setB.
        setA.
          builder
          10
        12
      0
```

Here we implemented the principles of the Builder pattern through measures supplied by the EO language. The builder object contains the subbuilder attribute object that implements the construction steps (setA, setB, setC, setName) as well as validation sub-steps (validateSide, validateName) and the finalize attribute that finishes the construction process and returns the resulting object. Initially, the instantiation of the builder object is substituted with a copy of the subbuilder object with an empty (meaning, all free attributes are unbound) copy of the constructing object. On each construction step, the subbuilder object returns itself by passing a changed version of the constructing object to its constructor. The construction steps have validation substeps that may implement some complex business logic. Validation steps may return an error or a validated object, which may lead to an interruption of the program execution and prevent inconsistency of the business logic. In conclusion, we would like to notice that the problem originally stated above (problem of optional configuration of objects with a lot of input parameters) and solved with the Builder pattern may not be actual to EO since it has the partial application mechanism that allows programmers to perform such configuration and, in addition, EO does not allow objects to have more than four free attributes (although, this restriction may be mitigated through passing complex object structures as free attributes). Nevertheless, the EO implementation of the Builder pattern may find its utilization in scopes where encapsulation of the creational process of objects is required. For instance, it may be needed when business logic validation of values passed for binding to free attributes is required.

5.7.11 Factory Method

The Factory Method Pattern is a creational object-oriented design pattern.

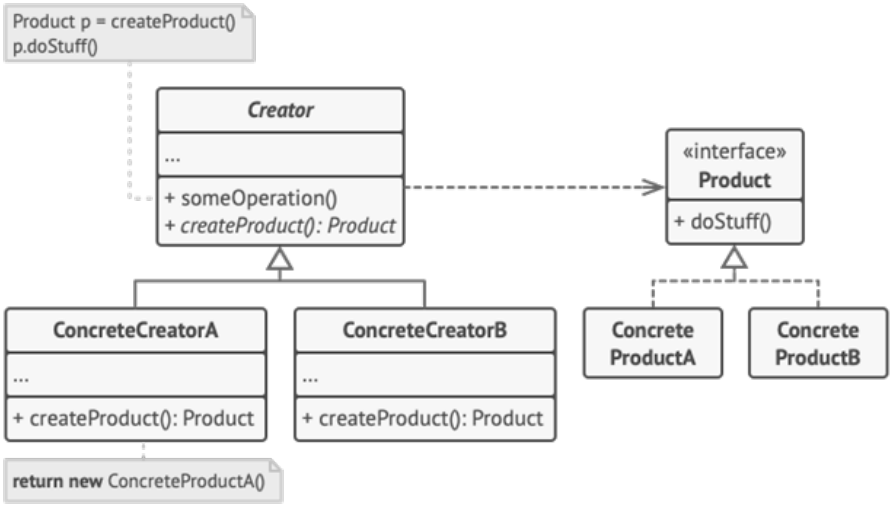

Figure 5.10: Factory Method Design Pattern

Purpose

Defines the creational method in the Creator superclass that defines a rule (that is, an interface or a contract) for creating an object (product) of some supertype Product. This method is used by the superclass or its more specific implementations, and the factory method can also be called from outside the class by other entities within the application. Concrete implementations of the Creator class with a factory method can return subtypes of the Product type, thereby ”tailoring” a specific implementation of the product class to the one required by the factory method contract. Hence, the pattern allows programmers to implement seamless configuration of the architecture of the application.

Problem

The pattern addresses the problem of extending the architecture of an application. By specifying the product contract (Product Interface) and by defining the class contract with the Factory Method Class, the architect separates the responsibility for creating the product itself from other methods of the creator class. This can be useful when:

1. It is not known what types of the Product class may be used in the future, but it may be appropriate to leave a headroom for a potential extension of the application architecture. Otherwise, this can be interpreted as an implementation of the ”Open / Closed” principle (O in SOLID).
2. Implementation of the principle of ”Single Responsibility” (S in SOLID). The code responsible for setting (configuring) a specific version of the product can be placed in a single place, for example, in a class that configures the application based on the environment settings. Here, the Dependency Injection mechanism can also be used to perform such a configuration in an invisible manner.
3. The pattern allows programmers to separate the logic of product creation from other logic of the creator class. This facilitates the reuse of identical code.

Structure

See fig. 5.10.

Code Instances Involved

Creator is an abstract class that defines the contract of the reutilized steps (here, it is someOperation) and the creational step (createProduct) of Products for its successors (concrete creators). Product is an interface for products (instances being created and configured through the Factory Method pattern). The interface defines the contract for all products so that these may be managed by the pattern.

Relation

Implementations of the Product interface are products. Inheritors of the Creator class provide concrete implementations for the creational method and inherit the rest methods. The concrete implementation of the creational method may return different implementations of Product, which implies the substitution of logic (or configurability of the application).

Implementation

The EO programming language does not have interfaces, classes, and types. Because of it, we may omit defining the Product interface contract (since it would not impose any requirements). Consider the following implementation of the pattern in EO:

```
[] > creator
  # left to be redefined
  [] > createObject
  # operation over products
  [] > performOperation
    createObject.getWeight.add 1 > @

[] > concreteCreatorA
  creator > @
  [] > createObject
    productA > @

[] > concreteCreatorB
  creator > @
  [] > createObject
    productB > @

[] > productA
  # let's suppose that this implementation
  # gets value from the production server
  [] > getWeight
    42 > @

[] > productB
  # let's suppose that this implementation
  # gets value from the testing server
  [] > getWeight
          24 > @
```

Here, we have the creator object with the performOperation attribute (the logic to be reused) and the createObject attribute (the logic to be redefined for flexible substitution of objects). The concreteCreatorA and concreteCreatorB objects have the creator object as their decoratee, so that these might inherit the reusable logic. Both objects define the createObject attribute that hides the original attribute of the same name from the decoration hierarchy. Objects productA and productB implement the attribute of interest (getWeight) differently. One of them may get the value from the production server, while another takes it from the testing environment. This example demonstrates the implementation of the classic version of the pattern in EO. However, we may consider a more EO-idiomatic example, free from additional structures (concrete creators) utilized in statically typed class-based object-oriented languages such as Java or C++. Consider the following example:

```
[@] > creator
  # operation over products
  [] > performOperation
    createObject.getWeight.add 1 > @

[] > productA
  # let's suppose that this implementation
  # gets value from the production server
  [] > getWeight
    42 > @

[] > productB
  # let's suppose that this implementation
  # gets value from the testing server
  [] > getWeight
    24 > @

[args...] > app
  creator > creatorObject
    []
      [] > createObject
        if. > @
          (args.get 0).eq "test"
          productB
          productA
```

Here, we used the technique of passing decoratee as a free attribute of the object creator. Its decoratee is passed in the app object. The decoratee has the only attribute createObject that the creator object inherits and relies on. The createObject attribute decides what version of a product should be chosen based on the environment configuration. This implementation of the pattern may be considered as more idiomatic and flexible from the EO perspective.

5.7.12 The Closures Functional Programming Technique

Since the EO programming language may be considered semi-functional, it might be useful to apply one of the widely adopted functional programming techniques, closures, in it. Simply put, the closures mechanism implies capturing outer lexical scope variables inside a function defined inside the scope with a consequent utilization of the function in other scopes. To support this technique, a language must operate over function as if they are first-class citizens (i.e., a language must return function or pass functions as parameters). Here is an example of this technique in JavaScript:

```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}
var add5 = makeAdder(5);
var add10 = makeAdder(10);
console.log(add5(2));  // 7
console.log(add10(2)); // 12
```

Here, we have the makeAdder function that returns an anonymous function capturing its outer state x. The state is then utilized when the returned function is applied with its own parameter y. In other words, the inner anonymous function remembers the value of x and uses it even when the actual value disappeared from the stack. This technique may be useful to emulate access modificators in functional languages:

```
var counter = (function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }

  return {
    increment: function() {
      changeBy(1);
    },

    decrement: function() {
      changeBy(-1);
    },

    value: function() {
      return privateCounter;
    }
  };
})();

console.log(counter.value());  // 0.

counter.increment();
counter.increment();
console.log(counter.value());  // 2.

counter.decrement();
console.log(counter.value());  // 1.
```

Here, the outer function counter returns a complex object-like structure containing functions that capture the state of the counter function. The state of the counter function is also complex: it has a mutable local variable privateCounter, and the changeBy function that mutates the value in the unified manner. The user code may not access the value and the mutating functions directly: both of them disappeared from the stack. However, the closures returned by the outer function still may do it. Hence, the technique allows functional programmers to simulate private state.

We surely may reproduce the similar technique of capturing the lexical scope in EO:

```
[] > counter
  memory 0 > privateCounter
  [val] > changeBy
    privateCounter.write > @
      privateCounter.add val
  [] > @
    [] > increment
      ^.^.changeBy 1 > @
    [] > decrement
      ^.^.changeBy (-1) > @
    [] > value
      ^.^.privateCounter > @
```

5.8 Conclusion

It is possible to conclude that (1) EO is principally applicable to all the considered patterns; (2) For some patterns, EO is able to give a fairly concise and intuitively clear code, since the language combines the features of Functional Programming and OOP; (3) the issues of effective implementation of patterns on EO are largely determined by the characteristics of the environment (IDE + compiler) and today remain open.

Also, EO has no local variables or any kind of stack-lifetime storage. Instead, any name refers to an object (stored in heap) that may be accessed through the scope of any other object via the dot-notation mechanism. Even anonymous objects may allow programmers to access its local scope (including parent and decoration hierarchies) freely. In addition, EO has no access modification instruments. This makes closures technique almost similar to the partial application mechanism. Moreover, the publicity of any attribute of any object makes encapsulation impossible in the language. This differentiates EO from functional programming languages and, also, from object-oriented languages. Absence of instruments of access modification (or simulation of it) may be a severe violation of object-oriented principle of encapsulation, which may lead to insecure environments breaking business logic of problem domains.

EO is fundamentally applicable to all the patterns considered. 2) For some patterns, EO is able to give a fairly concise and intuitively clear code, since the language combines the features of Functional Program (FP) and OOP. 3) the issues of effective implementation of patterns on EO are largely determined by the characteristics of the environment (IDE + compiler).

The implementation of the design patterns in Eolang is available at [69].

References

[1] John C. Reynolds. "GEDANKEN—a Simple Typeless Language Based on the Principle of Completeness and the Reference Concept". In: *Commun. ACM* 13.5 (May 1970), pp. 308–319. ISSN: 0001-0782. DOI: 10.1145/362349.362364. URL: https://doi.org/10.1145/362349.362364.

[2] W. P. Stevens, G. J. Myers, and L. L. Constantine. "Structured design". In: *IBM Systems Journal* 13.2 (June 1974), pp. 115–139. ISSN: 0018-8670. DOI: 10.1147/sj.132.0115. URL: https://doi.org/10.1147/sj.132.0115 (visited on 05/28/2021).

[3] G. Myers. "Software reliability - principles and practices". In: 1976.

[4] Meilir Page-Jones. *The practical guide to structured systems design: 2nd edition.* USA: Yourdon Press, 1988. ISBN: 978-0-13-690769-5.

[5] Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus.* USA, 1989.

[6] James M. Bieman and Byung-Kyoo Kang. "Cohesion and Reuse in an Object-Oriented System". In: *SIGSOFT Softw. Eng. Notes* 20.SI (Aug. 1995), pp. 259–262. ISSN: 0163-5948. DOI: 10.1145/223427.211856. URL: https://doi.org/10.1145/223427.211856.

[7] W. Zimmer. "Relationships between design patterns". en. In: *undefined* (1995). URL: https://www.semanticscholar.org/paper/Relationships-between-design-patterns-Zimmer/b7fd68d166ca62fc05fe267b69ac78c279c6ea4f (visited on 07/28/2021).

[8] B. Henderson-Sellers, L. Constantine, and I. Graham. "Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)". In: *Object Oriented Syst.* 3 (1996), pp. 143–158.

[9] Dirk Riehle and Heinz Züllighoven. "Understanding and using patterns in software development". en. In: *Theory and Practice of Object Systems* 2.1 (1996). ISSN: 1096-9942. DOI: 10.1002/(SICI)1096-9942(1996)2:1<3::AID-TAPO1>3.0.CO;2-#. (Visited on 08/02/2021).

[10] Douglas C. Schmidt, Mohamed Fayad, and Ralph E. Johnson. "Software patterns". In: *Communications of the ACM* 39.10 (Oct. 1996), pp. 37–39. ISSN: 0001-0782. DOI: 10.1145/236156.236164. URL: https://doi.org/10.1145/236156.236164 (visited on 08/02/2021).

[11] Wolfgang Pree and Hermann Sikora. "Design patterns for object-oriented software development (tutorial)". In: *Proceedings of the 19th international conference on Software engineering.* ICSE '97. New York, NY, USA: Association for Computing Machinery, May 1997, pp. 663–664. ISBN: 978-0-89791-914-2. DOI: 10.1145/253228.253810. URL: https://doi.org/10.1145/253228.253810 (visited on 08/01/2021).

[12] J. Coplien. "Software design patterns: common questions and answers". en. In: *undefined* (1998). URL: https://www.semanticscholar.org/paper/Software-design-patterns%3A-common-questions-and-Coplien/9544fccfd09a9315b29ced5bc1e69f572114b7ec (visited on 07/28/2021).

[13] J.K. Ousterhout. "Scripting: higher level programming for the 21st Century". In: *Computer* 31.3 (1998), pp. 23–30. DOI: 10.1109/2.660187.

[14] J. Bansiya et al. "A Class Cohesion Metric For Object-Oriented Designs". In: *J. Object Oriented Program.* 11 (1999), pp. 47–52.

[15] W. Schaffer and A. Zundorf. “Round-trip engineering with design patterns, UML, java and C++”. In: *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. ISSN: 0270-5257. May 1999, pp. 683–684. DOI: 10.1145/302405.302956.

[16] David J Barnes. *Object-Oriented Programming with Java: An Introduction*. 2000. URL: https://www.pearson.com/us/higher-education/program/Barnes-Object-Oriented-Programming-with-Java-An-Introduction/PGM26419.html.

[17] G. Antoniol et al. “Object-oriented design patterns recovery”. en. In: *Journal of Systems and Software* 59.2 (Nov. 2001), pp. 181–196. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(01)00061-9. URL: https://www.sciencedirect.com/science/article/pii/S0164121201000619 (visited on 07/29/2021).

[18] P. Wendorff. “Assessment of design patterns during software reengineering: lessons learned from a large commercial project”. In: *Proceedings Fifth European Conference on Software Maintenance and Reengineering*. Mar. 2001, pp. 77–84. DOI: 10.1109/CSMR.2001.914971.

[19] Michael Hahsler. *A Quantitative Study of the Application of Design Patterns in Java*. Feb. 2003.

[20] *Haskell Programming: Attractive Types*. 2003. URL: http://okmij.org/ftp/Haskell/types.html.

[21] Hirohisa Aman et al. “A Proposal of Class Cohesion Metrics Using Sizes of Cohesive Parts”. In: (Apr. 2004).

[22] Partha Kuchana. *Software Architecture Design Patterns in Java*. New York: Auerbach Publications, Apr. 2004. ISBN: 978-0-429-20970-3. DOI: 10.1201/9780203496213.

[23] Marek Vokac. “Defect Frequency and Design Patterns: An Empirical Study of Industrial Code”. In: *IEEE Transactions on Software Engineering* 30.12 (Dec. 2004), pp. 904–917. ISSN: 0098-5589. DOI: 10.1109/TSE.2004.99. URL: https://doi.org/10.1109/TSE.2004.99 (visited on 08/09/2021).

[24] D. Streitferdt, C. Heller, and I. Philippow. “Searching design patterns in source code”. In: *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*. Vol. 2. ISSN: 0730-3157. July 2005, 33–34 Vol. 1. DOI: 10.1109/COMPSAC.2005.135.

[25] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[26] Yaofei Chen and Rose Dios. “An Empirical Study of Programming Language Trends”. In: *IEEE Xplore* 8 (2006). ISSN: 2250-3153. DOI: 10.29322/IJSRP.8.12.2018.p8441. URL: http://www.ijsrp.org/research-paper-1218.php?rp=P848031.

[27] Steve Counsell, Stephen Swift, and Jason Crampton. “The Interpretation and Utility of Three Cohesion Metrics for Object-Oriented Design”. In: *ACM Trans. Softw. Eng. Methodol.* 15.2 (Apr. 2006), pp. 123–149. ISSN: 1049-331X. DOI: 10.1145/1131421.1131422. URL: https://doi.org/10.1145/1131421.1131422.

[28] Jeffrey Heer and Maneesh Agrawala. “Software Design Patterns for Information Visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (Sept. 2006). Conference Name: IEEE Transactions on Visualization and Computer Graphics, pp. 853–860. ISSN: 1941-0506. DOI: 10.1109/TVCG.2006.178.

[29] Lerina Aversano et al. “An empirical study on the evolution of design patterns”. In: *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ESEC-FSE '07. New York, NY, USA: Association for Computing Machinery, Sept. 2007, pp. 385–394. ISBN: 978-1-59593-811-4. DOI: 10.1145/1287624.1287680. URL: https://doi.org/10.1145/1287624.1287680 (visited on 07/29/2021).

[30] Sonate Company. *Maven: The Definitive Guide*. O'Reilly Media, Inc., 2008. URL: https://www.oreilly.com/library/view/maven-the-definitive/9780596517335/.

[31] Michael Kay. *XSLT 2.0 and XPath 2.0 Programmer's Reference, 4th Edition*. Michael Kay, 2008. URL: https://www.wiley.com/en-us/XSLT+2+0+and+XPath+2+0+Programmer%27s+Reference%2C+4th+Edition-p-9780470192740.

[32] Doug Tidwel. *XSLT, 2nd Edition*. O'Reilly Media, Inc, 2008. ISBN: 9780596527211. URL: https://www.oreilly.com/library/view/xslt-2nd-edition/9780596527211/.

[33] "A - Programming Languages Mentioned". In: *Programming Language Pragmatics (Third Edition)*. Ed. by Michael L. Scott. Third Edition. Boston: Morgan Kaufmann, 2009, pp. 819–829. ISBN: 978-0-12-374514-9. DOI: https://doi.org/10.1016/B978-0-12-374514-9.00028-8.

[34] J. Paquet and Serguei A. Mokhov. "Comparative Studies of Programming Languages; Course Lecture Notes". In: *ArXiv* abs/1007.2123 (2010).

[35] Aleksandar Bulajic and Slobodan Jovanovic. "An Approach to Reducing Complexity in Abstract Factory Design Pattern". In: *Journal of Emerging Trends in Computing and Information Sciences* 3.10 (2012). Publisher: Citeseer.

[36] Seyed Mohammad Hossein Hasheminejad and Saeed Jalili. "Design patterns selection: An automatic two-phase method". en. In: *Journal of Systems and Software*. Special issue with selected papers from the 23rd Brazilian Symposium on Software Engineering 85.2 (Feb. 2012), pp. 408–424. ISSN: 0164-1212. DOI: 10.1016/j.jss.2011.08.031. URL: https://www.sciencedirect.com/science/article/pii/S0164121211002317 (visited on 07/29/2021).

[37] Cheng Zhang and David Budgen. "What Do We Know about the Effectiveness of Software Design Patterns?" In: *IEEE Transactions on Software Engineering* 38.5 (Sept. 2012). Conference Name: IEEE Transactions on Software Engineering, pp. 1213–1231. ISSN: 1939-3520. DOI: 10.1109/TSE.2011.79.

[38] Marten Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, 2013.

[39] Mark Kraeling. "Chapter 7 - Embedded Software Programming and Implementation Guidelines". In: *Software Engineering for Embedded Systems*. Ed. by Robert Oshana and Mark Kraeling. Oxford: Newnes, 2013, pp. 183–204. ISBN: 978-0-12-415917-4. DOI: https://doi.org/10.1016/B978-0-12-415917-4.00007-4. URL: https://www.sciencedirect.com/science/article/pii/B9780124159174000074.

[40] Yegor Bugayenko. *OOP Alternative to Utility Classes*. May 2014. URL: https://www.yegor256.com/2014/05/05/oop-alternative-to-utility-classes.html.

[41] Raghuram Bharathan. *MApache Maven Cookbook*. Packt, 2015. URL: https://www.packtpub.com/product/apache-maven-cookbook/9781785286124.

[42] *Design Patterns | Set 1 (Introduction)*. en-us. Section: Design Pattern. Aug. 2015. URL: https://www.geeksforgeeks.org/design-patterns-set-1-introduction/ (visited on 07/23/2021).

[43] *Design Patterns | Set 2 (Factory Method)*. en-us. Section: Design Pattern. Aug. 2015. URL: https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/ (visited on 07/29/2021).

[44] Adnan Masood. *Learning F sharp Functional Data Structures and Algorithms*. 2015. URL: https://www.oreilly.com/library/view/learning-f-functional/9781783558476/.

[45] Michael L. Scott. *Programming Language Pragmatics, Fourth Edition*. Dec. 2015. URL: https://www.elsevier.com/books/programming-language-pragmatics/scott/978-0-12-410409-9.

[46] Robert V. Sebesta. *Concepts of programming languages*. 2015. URL: https://www.pearson.com/store/p/concepts-of-programming-languages/P100001149929/9780134997186.

[47] D. Alic, S. Omanovic, and V. Giedrimas. "Comparative analysis of functional and object-oriented programming". In: *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2016, pp. 667–672. DOI: 10.1109/MIPRO.2016.7522224.

[48] *google-java-format*. Google, 2016. URL: https://github.com/google/google-java-format (visited on 04/08/2021).

[49] *Abstract Factory Pattern*. en-us. Section: Design Pattern. July 2017. URL: https://www.geeksforgeeks.org/abstract-factory-pattern/ (visited on 07/29/2021).

[50] Yegor Bugayenko. *Elegant Objects*. 2017. URL: https://www.elegantobjects.org/.

[51] *Builder Design Pattern*. en-us. Section: Design Pattern. July 2017. URL: https://www.geeksforgeeks.org/builder-design-pattern/ (visited on 07/29/2021).

[52] Maryam Hooshyar Habib Izadkhah. *Class Cohesion Metrics for Software Engineering: A Critical Review*. 2017. URL: http://www.math.md/files/csjm/v25-n1/v25-n1-(pp44-74).pdf.

[53] Shahid Hussain, Jacky Keung, and Arif Ali Khan. "Software design patterns classification and selection using text categorization approach". en. In: *Applied Soft Computing* 58 (Sept. 2017), pp. 225–244. ISSN: 1568-4946. DOI: 10.1016/j.asoc.2017.04.043. URL: https://www.sciencedirect.com/science/article/pii/S1568494617302259 (visited on 07/28/2021).

[54] Dmitry Jemerov and Svetlana Isakova. *Kotlin in Action*. 2017. URL: https://www.manning.com/books/kotlin-in-action.

[55] Haiming Li, Qiyang Xia, and Yong Wang. "Research and Improvement of Kruskal Algorithm". en. In: *Journal of Computer and Communications* 5.12 (Sept. 2017). Number: 12 Publisher: Scientific Research Publishing, pp. 63–69. DOI: 10.4236/jcc.2017.512007. URL: http://www.scirp.org/Journal/Paperabs.aspx?paperid=79956 (visited on 08/28/2021).

[56] *Picocog*. GitHub, 2017. URL: https://github.com/ainslec/picocog (visited on 04/08/2021).

[57] G. Deepa et al. "Dijkstra Algorithm Application: Shortest Distance between Buildings". en-US. In: *International Journal of Engineering & Technology* 7.4.10 (Oct. 2018). Number: 4.10, pp. 974–976. ISSN: 2227-524X. DOI: 10.14419/ijet.v7i4.10.26638. URL: https://www.sciencepubco.com/index.php/ijet/article/view/26638 (visited on 08/28/2021).

[58] Myint Than Kyi and Lin Lin Naing. "Application of Ford-Fulkerson Algorithm to Maximum Flow in Water Distribution Pipeline Network". In: *International Journal of Scientific and Research Publications (IJSRP)* 8.12 (Dec. 2018). ISSN: 2250-3153. DOI: 10.29322/IJSRP.8.12.2018.p8441. URL: http://www.ijsrp.org/research-paper-1218.php?rp=P848031 (visited on 08/28/2021).

[59] Steven Zeil. *Dynamic Binding: Class-Appropriate behavior*. 2019. URL: https://www.cs.odu.edu/~zeil/cs330/latest/Public/dynamicBinding/index.html.

[60] Yegor Bugayenko. *Elegant objects*. 2020. URL: https://www.elegantobjects.org/.

[61] Yegor Bugayenko. "The Impact of Constructors on the Validity of Class Cohesion Metrics". In: *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. Mar. 2020, pp. 67–70. DOI: 10.1109/ICSA-C50368.2020.00021.

[62] *EO-lang compiler for simple cases*. HSE, Apr. 2021. URL: https://github.com/HSE-Eolang/eo (visited on 04/07/2021).

[63] *Plugin Developers Centre*. Maven, Apr. 2021. URL: https://maven.apache.org/plugin-developers/index.html (visited on 04/07/2021).

[64] HSE Team. *CI/CD*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/hse-eo-tests..

[65] HSE Team. *EO*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/eo.

[66] HSE Team. *EO Calculus*. en. HSE, 2021. URL: https://github.com/jizzel/jpeek/tree/master/src/main/java/org/jpeek/calculus/eo/EOCalculus.java.

[67] HSE Team. *EO CAMC*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/camc.eo.

[68] HSE Team. *EO CCM*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/CCM.eo.

[69] HSE Team. *Eo Design Patterns*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/eodesignpatterns.

[70] HSE Team. *Eo Graphs*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/eo_graphs.

[71] HSE Team. *EO JPeek*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/jpeek.

[72] HSE Team. *EO LCC*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/lcc.eo.

[73] HSE Team. *EO LCOM1*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/lcom1.eo.

[74] HSE Team. *EO LCOM1 1*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/lcom1_1.eo.

[75] HSE Team. *EO LCOM2*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/lcom2.eo.

[76] HSE Team. *EO LCOM3*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/lcom3.eo.

[77] HSE Team. *EO LCOM5*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/lcom5.eo.

[78] HSE Team. *EO NHD*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/nhd.eo.

[79] HSE Team. *EO OCC*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/occ.eo.

[80] HSE Team. *EO PCC*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/pcc.eo.

[81] HSE Team. *EO SCOM*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/scom.eo.

[82] HSE Team. *EO Skeleton*. en. HSE, 2021. URL: https://github.com/jizzel/jpeek/tree/master/src/main/java/org/jpeek/skeleton/eo/EOSkeleton.java.

[83] HSE Team. *EO TCC*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/jpeek/blob/master/src/eo/tcc.eo.

[84] HSE Team. *Factorial example*. HSE, Apr. 2021. URL: https://github.com/HSE-Eolang/eo/blob/master/sandbox/eo/app.eo (visited on 04/07/2021).

[85] HSE Team. *Fibonacci example*. HSE, Apr. 2021. URL: https://github.com/HSE-Eolang/eo/blob/master/sandbox/eo/fibonacci.eo (visited on 04/07/2021).

[86] HSE Team. *Ford Falkerson*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/eo_graphs/tree/master/src/main/eo/fordfalkerson.

[87] HSE Team. *Kruskal*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/eo_graphs/blob/master/src/main/eo/kruskal.eo.

[88] HSE Team. *Pi digits example*. HSE, Apr. 2021. URL: https://github.com/HSE-Eolang/sandbox-examples/blob/main/eo/pi.eo (visited on 04/07/2021).

[89] HSE Team. *Prim*. en. HSE, 2021. URL: https://github.com/HSE-Eolang/eo_graphs/blob/master/src/main/eo/prim.eo.

[90] HSE Team. *Sum of array example*. HSE, Apr. 2021. URL: https://github.com/HSE-Eolang/sandbox-examples/blob/main/eo/sum.eo (visited on 04/07/2021).

[91] "A catalogue of general-purpose software design patterns". In: pp. 330–339. DOI: 10.1109/TOOLS.1997.654742.

[92] *Abstract Factory*. en. URL: https://refactoring.guru/design-patterns/abstract-factory (visited on 07/29/2021).

[93] *Abstract Factory Design Pattern*. en-US. URL: https://springframework.guru/gang-of-four-design-patterns/abstract-factory-design-pattern/ (visited on 07/29/2021).

[94] Marat Akhin and Mikhail Belyaev. *Kotlin language specification*. URL: https://kotlinlang.org/spec/introduction.html.

[95] aventador3000. *Node Packages*. URL: https://github.com/aventador3000/Node_package.

[96] Yegor Bugayenko. *Elegant Objects*. URL: https://www.elegantobjects.org/.

[97] *Comparing Programming Languages*. URL: https://www.cs.ucf.edu/~leavens/ComS541Fall98/hw-pages/comparing/ (visited on 08/30/2021).

[98] *Design Pattern - Abstract Factory Pattern - Tutorialspoint*. URL: https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm (visited on 07/29/2021).

[99] *Design Pattern - Factory Pattern - Tutorialspoint*. URL: https://www.tutorialspoint.com/design_pattern/factory_pattern.htm (visited on 07/29/2021).

[100] *Design Patterns - Adapter Pattern - Tutorialspoint*. URL: https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm (visited on 08/01/2021).

[101] *Design Patterns - Builder Pattern - Tutorialspoint*. URL: https://www.tutorialspoint.com/design_pattern/builder_pattern.htm (visited on 07/29/2021).

[102] *Design Patterns and Refactoring*. en. URL: https://sourcemaking.com (visited on 07/23/2021).

[103] *Design Patterns: Elements of Reusable Object-Oriented Software [Book]*. en. ISBN: 9780201633610. URL: https://www.oreilly.com/library/view/design-patterns-elements/0201633612/ (visited on 08/02/2021).

[104] *Design Patterns: Factory Method in C++*. en. URL: https://refactoring.guru/design-patterns/factory-method/cpp/example (visited on 07/29/2021).

[105] *Factory Method*. en. URL: https://refactoring.guru/design-patterns/factory-method (visited on 07/29/2021).

[106] Luis Fernández and Rosalía Peña. "A Sensitive Metric of Class Cohesion". In: *Information Theories and Applications* 13 (), p. 2006.

[107] *Huston Design Patterns*. URL: http://www.vincehuston.org/dp/ (visited on 07/25/2021).

[108] Chidamber & Kemerer. *Project Metrics Help - Chidamber & Kemerer object-oriented metrics suite*. URL: https://www.aivosto.com/project/help/pm-oo-ck.html (visited on 06/11/2021).

[109] Mitendra Mahto On. *Code simplicity: A language independent perspective | Hacker Noon*. en. URL: https://hackernoon.com/code-simplicity-a-language-independent-perspective-756cf031c913 (visited on 08/30/2021).

[110] Mohammed T. Abo Alroos. *Software Metrics*. 20XX. URL: http://site.iugaza.edu.ps/mroos/files/Software-Metrics1.pdf.

Appendix A

Appendix

A.1 Fibonacci Example

A.1.1 app.eo

```
+package sandbox
+alias fibonacci sandbox.fibonacci
+alias stdout org.eolang.io.stdout
+alias sprintf org.eolang.txt.sprintf

[args...] > app
  sprintf > @
    "%dth Fibonacci number is %d\n"
    (args.get 0).toInt > n
    fibonacci n
```

A.1.2 fibonacci.eo

```
    +package sandbox

[n] > fibonacci
  if. > @
    n.less 3
    small n
    rec n 1 1

  [n] > small
    if. > @
      n.eq 2
      1
      n

  [n minus1 minus2] > rec
    if. > @
      n.eq 3
      minus1.add minus2
      rec (n.sub 1) (minus1.add minus2) minus1
```

A.1.3 EOapp.java

```
package sandbox;

import org.eolang.*;
import org.eolang.core.*;

/** Package-scope object 'app'. */
public class EOapp extends EOObject {

  /** Field for storing the 'args' variable-length free attribute. */
  private final EOarray EOargs;

  /**
   * Constructs (via one-time-full application) the package-scope object 'app'.
   *
   * @param EOargs the object to bind to the 'args' variable-length free attribute.
```

```
   */
  public EOapp(EOObject... EOargs) {
    this.EOargs = new EOarray(EOargs);
  }

  /** The decoratee of this object. */
  @Override
  public EOObject _getDecoratedObject() {
    return new org.eolang.txt.EOsprintf(
        new EOstring("%dth Fibonacci number is %d\n"),
        this.EOn(),
        new sandbox.EOfibonacci(this.EOn()));
  }

  /** Returns the object bound to the 'args' input attribute. */
  public EOarray EOargs() {
    return this.EOargs;
  }

  /** Application-based bound attribute object 'n' */
  public EOObject EOn() {
    return ((this.EOargs())._getAttribute("EOget", new EOint(0L)))._getAttribute("EOtoInt");
  }
}
```

A.1.4 EOfibonacci.java

```
package sandbox;

import org.eolang.*;
import org.eolang.core.*;

/** Package-scope object 'fibonacci'. */
public class EOfibonacci extends EOObject {

  /** Field for storing the 'n' free attribute. */
  private final EOObject EOn;

  /**
   * Constructs (via one-time-full application) the package-scope object 'fibonacci'.
   *
   * @param EOn the object to bind to the 'n' free attribute.
   */
  public EOfibonacci(EOObject EOn) {
    this.EOn = EOn;
  }

  /** The decoratee of this object. */
  @Override
  public EOObject _getDecoratedObject() {
    return ((this.EOn())._getAttribute("EOless", new EOint(3L)))
        ._getAttribute(
            "EOif", this.EOsmall(this.EOn()), this.EOrec(this.EOn(), new EOint(1L), new EOint(1L)));
  }

  /** Returns the object bound to the 'n' input attribute. */
  public EOObject EOn() {
    return this.EOn;
  }

  /** Abstraction-based bound attribute object 'small' */
  public EOObject EOsmall(EOObject EOn) {
    return new EOsmall(EOn);
  }

  /** Abstraction-based bound attribute object 'rec' */
  public EOObject EOrec(EOObject EOn, EOObject EOminus1, EOObject EOminus2) {
    return new EOrec(EOn, EOminus1, EOminus2);
  }

  /** Attribute object 'small' of the package-scope object 'fibonacci'. */
  private class EOsmall extends EOObject {
```

```
  /** Field for storing the 'n' free attribute. */
  private final EOObject EOn;

  /**
   * Constructs (via one-time-full application) the attribute object 'small' of the package-scope
   * object 'fibonacci'.
   *
   * @param EOn the object to bind to the 'n' free attribute.
   */
  public EOsmall(EOObject EOn) {
    this.EOn = EOn;
  }

  /** Returns the parent object 'fibonacci' of this object */
  @Override
  public EOObject _getParentObject() {
    return EOfibonacci.this;
  }

  /** The decoratee of this object. */
  @Override
  public EOObject _getDecoratedObject() {
    return ((this.EOn())._getAttribute("EOeq", new EOint(2L)))
        ._getAttribute("EOif", new EOint(1L), this.EOn());
  }

  /** Returns the object bound to the 'n' input attribute. */
  public EOObject EOn() {
    return this.EOn;
  }
}

/** Attribute object 'rec' of the package-scope object 'fibonacci'. */
private class EOrec extends EOObject {

  /** Field for storing the 'n' free attribute. */
  private final EOObject EOn;
  /** Field for storing the 'minus1' free attribute. */
  private final EOObject EOminus1;
  /** Field for storing the 'minus2' free attribute. */
  private final EOObject EOminus2;

  /**
   * Constructs (via one-time-full application) the attribute object 'rec' of the package-scope
   * object 'fibonacci'.
   *
   * @param EOn the object to bind to the 'n' free attribute.
   * @param EOminus1 the object to bind to the 'minus1' free attribute.
   * @param EOminus2 the object to bind to the 'minus2' free attribute.
   */
  public EOrec(EOObject EOn, EOObject EOminus1, EOObject EOminus2) {
    this.EOn = EOn;
    this.EOminus1 = EOminus1;
    this.EOminus2 = EOminus2;
  }

  /** Returns the parent object 'fibonacci' of this object */
  @Override
  public EOObject _getParentObject() {
    return EOfibonacci.this;
  }

  /** The decoratee of this object. */
  @Override
  public EOObject _getDecoratedObject() {
    return ((this.EOn())._getAttribute("EOeq", new EOint(3L)))
        ._getAttribute(
            "EOif",
            (this.EOminus1())._getAttribute("EOadd", this.EOminus2()),
            new EOrec(
                (this.EOn())._getAttribute("EOsub", new EOint(1L)),
                (this.EOminus1())._getAttribute("EOadd", this.EOminus2()),
                this.EOminus1()));
```

```
    }

    /** Returns the object bound to the 'n' input attribute. */
    public EOObject EOn() {
      return this.EOn;
    }

    /** Returns the object bound to the 'minus1' input attribute. */
    public EOObject EOminus1() {
      return this.EOminus1;
    }

    /** Returns the object bound to the 'minus2' input attribute. */
    public EOObject EOminus2() {
      return this.EOminus2;
    }
  }
}
```