ESEM

Authors Here
Faculty of Computer Science
Higher School of Economics
Moscow, Russia
empopov@edu.hse.ru

Abstract—the EO programming language is a novel initiative that aims to drive the course of elaboration of the proper practical application of the object-oriented programming paradigm. The EO language follows the eponymous philosophy advocating pure objects, free from incorrectly made design decisions common for mainstream technologies. EO bases on the formal model of object calculus—the φ-calculus. This work aims to enhance the implementation of the transpiler and the standard object library of the EO programming language. This study highlights essential features of the language and the underlying formal calculus and proposes a translation scheme of EO programs to Java source code with comparison of this scheme to an existing solution. The findings of this work are the proposed transcompilation model and a renewed transpiler implementing it. This work is valuable for the EO project as it facilitates the language to evolve and prepares it for further assessments on practical applicability, interoperability with Java, performance in enterprise applications, et alia.

Keywords—object-oriented programming, Java, the EO programming language, transcompilation, transpiler, the Phi calculus, elegant objects

I. Introduction

Object-oriented programming (OOP) has been the dominant paradigm in the software development industry for the past two decades. Many technological enterprises and leading digital solution providers that utilize the mainstream OOP languages suffer from a lack of quality of their projects despite the tremendous effort and resources that have been invested in their development. Quality and maintainability issues might be explained by the essence of inherent flaws in the design of the programming languages and the OOP paradigm itself, as industry experts point out. Thus, drastically new languages and approaches have been developing.

The EO programming language is one of the promising technologies that has arisen to drive the course of elaboration of the proper practical application of the OOP paradigm. The EO language emerged from the "Elegant Objects" philosophy that advocates the vision of pure OOP programming, free from the incorrectly made design decisions common for the mainstream technologies. These are static code entities, inheritance, classes, mutable objects, null references, reflection, and global variables, amongst others. EO is based on the formal model of object calculus—the φ-calculus. The calculus defines four elemental operations as sufficient to describe object-oriented programming paradigm properties and semantics.

The aim of this work is to enhance the implementation of the transpiler and the standard object library of the EO programming language. Objectives of this paper are to analyze the language and the φ-calculus identifying essential parts of EO, to propose a transcompilation model of EO programs to Java source code, to compare the proposed model to an existing scheme, to probe the proposed translation model through implementation of a renewed EO transpiler and the standard object library of the language.

The findings of this work are the proposed transcompilation model and a renewed transpiler implementing it. The EO programming language is a research and development project that remains in an undeveloped state. Hence, this work is considered valuable for the language to evolve and be prepared for further assessments on practical applicability, interoperability with Java, performance in enterprise applications, and other analyses and improvements.

This paper is structured as follows. Section II describes the EO language and its bases and summarizes the elemental parts of the language. Section III and IV depict the existing and proposed transcompilation models of EO programs to Java source code qualitatively. Then, Section V outlines the implementation details of the transpiler based on the proposed model and compares performance of the new and the existing solutions quantitatively. Section VI provides a discussion of the findings of this work with a resume of advantages and limitations of the proposed model. Finally, Section VII concludes this paper and projects future work directions based on this research.

II. The EO Programming Language

This section provides a concise description of the object of this study—the EO programming language. The description covers the Elegant Objects philosophy, the Phi calculus, the elementals of the EO language, syntax and semantics of EO, and its relation to programming languages and paradigms.

A. The Elegant Objects Philosophy

The EO language emerged from the "Elegant Objects" philosophy which is a congregation of advised conventions to achieve proper object-oriented principles and designs in practice [2, 3]. The "Elegant Objects" philosophy advocates code-free constructors, immutable objects, design-by-contract programming and does not admit null references, static code entities, reflective programming, type introspection at runtime, the pattern of getters and setters, inheritance mechanism, and others. The EO language—which is an attempt to form a small OOP language based on the philosophy—abides by most of the principles as it will be shown in more detail in section C.

B. The Phi Calculus

The φ-calculus is a model of object calculus formulated by Y. Bugayenko as a proposition of the formal basis for the object-oriented programming paradigm and the EO language [1]. The φ-calculus vastly relies on the λ-calculus, applied in functional programming, as it defines objects—first-class entities of the model, in contrary to the λ-calculus—as sets of sub-objects, and the internals of atomic objects (meaning, non-reducible objects with the implementation details defined outside of the φ-calculus) as lambda terms. The φ-calculus defines abstraction, application, decoration, and dataization as the principal elements that comprise the sufficient set of

operations that can describe object-oriented programming paradigm properties and semantics.

The Phi and Lambda calculi are nearly cognate structurally and terminologically. The abstraction operation in either of the models is used to define new entities with input arguments and internal structures. In the λ-calculus, a function declared through abstraction can have only one argument and one internal lambda term determining the expression that the function reduces to [5]. In contrast, objects declared by means of the operation in the φ-calculus can have any number of free attributes (arguments or inputs) and bound attributes (outputs or objects structurally associated with their parent object).

Similarly, the operation of application in both calculi is used to substitute input arguments of entities. In the λ-calculus, only one term can be applied to another (although, currying technique enables functions to have multiple arguments [8, Sec. 2.2.1]). In the φ-calculus, an arbitrary number of terms can be applied to an object being instantiated through application. Due to the hierarchical nature of objects, the φ-calculus additionally defines the dot-notation mechanism that allows application terms to access attributes of objects (including special cases, namely, parent objects, self-referencing, and decorated objects). The φ-calculus adapts the partial application technique used in functional programming. This technique enables application terms to bind only a subset of arguments to free attributes of an object being instantiated, leaving some of attributes unbound.

The operation of object decoration defined in the φ-calculus has no direct counterpart in the λ-calculus. Nevertheless, it may be compared with the function composition mechanism widely employed in functional programming [7, Sec. 4.5.2]. As the function composition technique facilitates code factoring in functional programming paradigm, so does the decoration operation in the φ-calculus. The operation allows an object to refer to another object— either through abstraction or application—as to the object it decorates. The instantiated link between the objects extends the set of attributes of the decorator object to the union of the initial set of its attributes and the set of attributes of the decoratee object. Hence, the decoration operation performs the construction of the eponymous pattern utilized in object-oriented programming [9, Ch. 4]. Thus, decoration allows objects to be extended or, oppositely, factored to maintainable parts.

The dataization operation performs the evaluation strategy over objects to extract data they represent. Dataization may be compared to the reduction strategy defined in the λ-calculus. While reduction operations accomplish a sequence of lambda terms substitutions, dataization, on contrary, performs recursive object tree evaluation due to the hierarchical structure of objects and, hence, applications of them. The dataization operation is declared as a call-by-need operation. Therefore, the evaluation strategy of the φ-calculus is lazy in its nature.

Both Phi and Lambda calculi do not define a type system as a part of them leaving it to applications utilizing them—programming languages and compilers. A plain and simple implementation of the φ-calculus would have a dynamical type system due to the hierarchical multi-faceted nature of objects causing need to determine existence of referenced attributes of an object in runtime.

C. Fundamental Elements, Syntax, and Semantics of EO

This section covers the elemental parts, syntactical and semantical properties of the EO programming language [4]. These include objects, attributes, the four operations defined in the φ-calculus, and the type system.

1) Objects and Attributes

An object—the centric notion of EO—is a set of attributes. Every entity in the EO language is an object. An object can have an arbitrary number of free and bound attributes. Attributes of objects—and, hence, objects as well—are immutable, meaning attributes can be associated with corresponding objects only once, and no modifications are allowed. Objects with at least one free attribute are abstract, and those with no free attributes are closed.

Every object has a scope it belongs to. An object may be scoped to a package (package-level scope), to another object declared through abstraction (attribute-level scope), or to an application term (application-level or anonymous scope). Scope is declared structurally (i.e., by a context an object is declared in) and cannot be changed dynamically. An object accesses other objects through its scope. There are four types of object access in EO:

- Accessing the parent object. This type of access is always explicit only, so to reference the parent object or an attribute of it a programmer must use the special "^" identifier.
- Accessing objects in the decoration hierarchy. This type of access is implicit by default and may be denoted unambiguously through the special "@" identifier. Due to semantics of decoration (see Decoration section below), an implicitly referenced attribute in the decoration hierarchy may be shadowed by an attribute with a similar name defined in the object itself.
- Accessing the object itself or one of its attributes. This type of access is implicit by default and may be denoted unambiguously through the special "$" identifier.
- Accessing an object from an outer scope (package-level objects referencing). If an accessed object is declared outside of the file where the reference appears, it is accessed through the alias name given on the file level.

Attribute access in any scope is performed through the dot-notation.

2) Abstraction and Application

Abstraction and application allow a programmer create objects. However, they operate differently. Structurally new objects can be declared through abstraction. In other words, definition of free attributes as well as the internal structure of an object is possible by means of abstraction only. Application is used to instantiate an object with binding arguments to its free attributes.

Both operations are hierarchical and recursive. Abstraction may define an arbitrary number of inner attribute objects based on abstraction and application, and one application term may have other applications (and anonymous abstractions) inside it. This property, thus, makes EO programs structurally (or declaratively) hierarchical.

Both operations have special semantical and syntactical features. Either operation can produce named or anonymous

objects. Anonymous abstraction may be in-lined (meaning, expressed in one line). Application syntax may be vertical (inverted) or horizontal (plain).

3) Decoration

The decoration operation defined in the φ-calculus can be performed over an object in a declarative manner through binding its special "@" attribute to a language expression (either abstraction or application) denoting a decorated object. A decorator object inherits all the attributes of its decoratee and may define its own attributes including those that shadow some of the attributes of the decorated object. Shadowing is a semantical property of the language that defines rules of resolving names when similar identifiers are declared in different scopes. Attributes of a decorator object shadow those of a decorated object when their names and argument signatures are identical.

4) Dataization

As described in section B, dataization defines the evaluation scheme of EO programs. For all programmer-defined objects, dataization relies on corresponding decorated objects entirely. To put it more simply, objects defined in EO programs delegate their evaluation scheme to their decoratees. In contrast, atomic objects (i.e., defined and implemented outside of the EO environments, for instance, objects of the standard library) may declare their own evaluation strategies.

This property of dataization implies several notable characteristics of the language. First, due to lazy nature of the decoration operation defined in the φ-calculus, all programmer-defined objects have lazy evaluation strategy as well. However, atomic objects may control their evaluation strategy freely making it eager or lazy in different contexts. Second, since dataization is the evaluation mechanism of EO and it relies on decoration completely, a program written in EO is decomposed to a set of objects, and an entry point object of the program defines its evaluation path through decoration by binding other objects to the "@" attribute (while these denote their evaluation schemes in the same manner). Hence, EO programs are evaluated hierarchically, too. In fact, an overall dataization strategy of any object or a set of objects may be denoted structurally as a general tree (meaning, each node may have an arbitrary number of child nodes). Moreover, some of objects in the evaluation tree must delegate their dataization scheme to atomic objects to make the tree reducible. Finally, programmer-defined objects with no decoratees may not be evaluated.

5) Type System

The EO programming language has a dynamic type system since it has no explicit type declaration syntax and due to the hierarchical multi-faceted nature of objects causing need to determine existence of referenced attributes of an object in runtime. However, types of objects may be inferred in some cases—when a type of an object may be determined directly from the source code—and, hence, type checking may take place. An example of non-determinant typing context (where a type may not be inferred) is referencing attributes of free attributes of an object. Semantically, free attributes of objects must be typed dynamically to keep the flexible nature of the application operation.

D. Relation of EO to Programming Languages and Paradigms

The EO programming language bases on the novel formal model of the φ-calculus showing resemblance with its functional programming equivalent—the λ-calculus—and follows the principals of the "Elegant Objects" philosophy that renounces traditional OOP techniques. EO is not a functional language since it has objects, not functions, as its first-class citizens. The φ-calculus rather supersets and redefines the Lambda calculus in terms of objects, that is why the language may not be classified as functional. On the other hand, "Elegant Objects" principles backing the language shrinks commonly used OOP features to a restricted subset of them. This obstructs classifying the EO language to the object-oriented programming paradigm in its commonly appreciated vision directly as well. However, EO may be categorized as a declarative object-oriented programming as it operates over objects—its primary and only entities—declaratively.

//TODO: add syntax to appendix

//TODO: comparison table

III. The Existing Implementation of the Language

This section describes the transcompilation model of EO programs to Java source code (as well as its implementation in a form of a transpiler) proposed by Yegor Bugayenko, the main contributor of the EO programming language project. Further in this section, the model is referred to as existing or current.

A. The Transcompilation Model

The existing model defines a transcompilation scheme of EO programs to Java source code. This subsection summarizes the mapping rules for each elemental part of the language defined within the scheme.

1) Objects Definition Through Abstraction

Objects declared in EO programs by means of the abstraction operation are mapped to Java classes extending the "PhDefault" base type that defines the standard internal structure of all objects in the EO runtime environment, specifically the mechanism of objects cloning and the apparatuses of storing, retrieving, and mutating attributes of objects.

Attributes of an object are stored within an associative array inside the object. Keys of elements of the array are names of the attributes of the object, while values are instances of classes of the EO standard library denoting the contents of the attributes—corresponding EO objects. Target Java classes resulted from translation of EO objects declared through abstraction contain the constructor that assembles the associative array defining free and bound attributes of corresponding EO objects. A free attribute of an object is stored as an instance of the "AtFree" EO runtime standard class (meaning, the attribute may be substituted with a concrete value) and a bound attribute is stored as an instance of the "AtBound" class containing a lambda expression that defines the contents of the bound attribute.

The associative array containing the attributes of the object may be mutated through the operation of addition of new attributes. Due to the mutable nature of the array, the "PhDefault" base type defines the standard cloning utility method that instantiates an exact copy of the object. This

method is used to ensure the characteristic of immutability of objects when they are copied or passed as arguments through the operation of application.

As described in Section II (subsection C, paragraph 1), every object is declared in one of the scopes:

- Package-level scope.
- Attribute-level scope.
- Application-scope (or anonymous scope).

In any case, objects declared through abstraction are translated to public package-level Java classes, and each Java class is stored in a separate file. Every separated file is kept within a Java package with a name identical to the one declared through the "package" meta directive at the top of the EO source file the abstracted object is stored in. Nested and anonymous EO objects are flattened out and stored outside the original scopes they belong to. To mitigate naming conflicts that may be potentially caused due to the flattened nature of the target source code, the existing model encodes the original scopes of objects in names of their Java files and classes delimiting parts of the names with a dollar sign symbol.

2) Objects Instantiation Through Application

An application term—that instantiates an object providing its free attributes with concrete values—associated with a bound attribute of some object is translated to a Java lambda expression contained in an instance of the "AtBound" class stored in the attributes associative array of the object. This lambda expression contains statements denoting the actual contents of the application term. The application term, as described in Section II (subsection C, paragraph 2) may hierarchically include other application terms. These are translated as Java statements placed within the same lambda expression.

The Java statements that the application term is translated may consist of the following parts:

- Class instantiation. This part is used when a package-level object is applied. The instantiated object is then copied and (optionally) its free attributes are provided with concrete values.
- Attribute access through creation of an instance of the "PhMethod" class. This part is used when an attribute of an object is applied (including special attributes "@", "^", "$", as described in Section II (subsection C, paragraph 1)).
- Free attribute binding through creation of an instance of the "PhWith" class. This part is used when free attributes of an object are provided with concrete values through the application term.
- Object cloning through creation of an instance of the "PhCopy" class. This part is used to ensure the immutability characteristic of EO objects when mutating attributes.

3) Objects Decoration

As described in Section II (subsection C, paragraph 3), decoration in EO is performed in a declarative manner through binding the special "@" attribute of a decorator object to a language expression (either abstraction or application) denoting a decorated object. Hence, internally, the constructor of the target Java class of the decorator object appends a new element to the attributes associative array of the class. The appended element has the key "φ" and a value that corresponds to the decorated object.

4) Dataization Strategy of Objects

Dataization of EO objects defined by programmer relies on the decoration operation. Transpiled Java source code does not perform any actions unless it is demanded since the actual code is placed within lambda statements. Once dataization of an object is started, the evaluation tree is built and traversed down to atomic objects defined in the standard library. The standard EO objects are lazy. Hence, dataization strategy of all objects is lazy.

5) Typization

Attribute access of objects is done through the "PhMethod" class instantiation. This class performs attribute access through a lookup in the attribute associative array of the object. If the object does not have the referenced attribute, the evaluation of the program fails. Otherwise, the referenced attribute is returned.

Free attribute binding is done through the "PhWith" instance creation. This class performs a lookup of the referenced free attribute in the attribute associative array of the object. If the object does not have the referenced attribute, the program fails. Otherwise, the referenced attribute is bound.

All objects in the EO runtime environments have the same Java type. From the Java Virtual Machine perspective, all of the objects have the same type with identical set of fields, methods and constructors. Hence, no compile-time type checks are performed. As showed above, all verifications are done at runtime through lookups of the dynamically formed attributes associative array. Thus, the existing model has a dynamic type system.

B. The Implementation of the Model

C. Limitations and Problems of the Existing Model

IV. The Proposed Transcompilation Model

This section describes the transcompilation model of EO programs to Java source code (as well as its implementation in a form of a transpiler) proposed in this paper. Further in this section, the model is referred to as the proposed model.

A. The Transcompilation Model

1) Objects Definition Through Abstraction

Objects declared in EO programs by means of the abstraction operation are mapped to Java classes extending the "EOObject" base type that defines the standard internal structure of all objects in the EO runtime environment, specifically the mechanism of retrieving data and attributes of objects.

An EO object of the package-level scope is translated into a Java class stored in a separate file. EO objects of attribute-level and anonymous scopes are not present as separate Java files. Instead, these are put to the scopes they are originated from. So, abstraction-based attributes of an object are translated to private inner classes [10], and anonymous EO objects are translated as in-place local Java classes placed directly within methods they are used in [11]. This technique makes target code more dense, safe and delegates management of scoping and parent hierarchies to the Java Virtual Machine.

The target Java class (of any scope) has only one constructor. There are two possible cases:

1. If the source EO object has no free attributes, the resulting constructor has no arguments.
2. If the source EO object has free attributes, the resulting constructor has all the arguments in the order of their appearance in the source program. All arguments of the constructor are of type "EOObject". In the case if free attributes are present, the default constructor is disabled (which is the default Java semantics feature).

For each free attribute of the source EO object, the following target Java entities are generated:

1. A private final class field of type "EOObject".
2. A public wrapper method of return type "EOObject". The method returns the corresponding field.
3. An argument in the sole constructor (in the order of appearance of the free attribute in the source EO object). The body of the constructor sets the corresponding field to the value of the argument.

Bound attributes of the source EO object are translated as follows:

1. For every bound attribute, a wrapper method of the "EOObject" return type.
2. If the bound attribute is constructed through the application operation in the source EO program, then the target Java code denoting the application term is placed into the wrapper method.
3. If the bound attribute is constructed through the abstraction operation in the source EO program, then a private inner class is generated. The translation scheme for attribute objects is the same, except an overridden version of the "_getParentObject" method is generated. This method returns a reference to the parent object of the attribute object. The wrapper method returns a new instance of the generated inner class passing arguments (free attributes) to it in the order of their appearance in the source EO program.

The proposed model is fully immutable since a code fragment that applies (or copies) a class can access one and the only constructor and all arguments of the constructor are required. Fields that store free attributes of the object are final. The default constructor is disabled. Once sat, an attribute cannot be changed. Bound attributes are translated to methods (and, in some cases, inner classes). These Java source code entities cannot be changed. Hence, objects are fully immutable without proposing cloning techniques utilized in the existing model.

2) Objects Instantiation Through Application

An application term—that instantiates an object providing its free attributes with concrete values—associated with a bound attribute of some object is translated to a Java expression placed in the corresponding wrapper method. The application term, as described in Section II (subsection C, paragraph 2) may hierarchically include other application terms. These are translated to subexpressions contained within the parent expression.

The Java expression that the application term is translated to may consist of the following parts:

- Constructor call. This part is used when a package-level object is applied. Objects to bind to free attributes of the object are passed to the constructor.
- Attribute access through the "_getAttribute" method call (or through a plain Java method call in known contexts). This part is used when an attribute of an object is applied. Objects to bind to free attributes of the attribute object are passed to the method.

All arguments to constructor and method calls are wrapped with an instance of the special class "EOThunk". The actual argument is stored in a lambda expression inside the thunk object. The thunk object unwraps its contents once any of the messages is sent to it (in other words, it implements the call-by-need evaluation scheme). This technique is used to avoid the eager evaluation scheme of arguments—that contradicts the semantics of EO—embedded into Java.

3) Objects Decoration

As described in Section II (subsection C, paragraph 3), decoration in EO is performed in a declarative manner through binding the special "@" attribute of a decorator object to a language expression (either abstraction or application) denoting a decorated object. Hence, the decoration operation is translated in the same way as all bound attributes, except an overridden method "_decoratee" is generated as the wrapper.

4) Dataization Strategy of Objects

Dataization of EO objects defined by programmer relies on the decoration operation. Transpiled Java source code does not perform any actions unless it is demanded since the actual code is placed within lambda statements. Once dataization of an object is started, the evaluation tree is built and traversed down to atomic objects defined in the standard library. Some of the standard EO objects are eager (specifically, ones that perform arithmetical computations). Hence, dataization strategy of the proposed model depends on the objects used.

5) Typization

Attribute access of objects is done through the "_getAttribute" method that utilizes the Java Reflection API. The API dynamically lookups the list of methods of the object. If the object does not have the referenced method, the evaluation of the program fails. Otherwise, the referenced object is returned.

Free attribute binding is done through passing arguments to constructors or methods of classes. If the call signature does not correspond to the one declared in the callee, the program fails. However, this verification is rather synthetical since it does not check if the passed objects would match the callee internal structure semantically. In the case when the object being instantiated is an attribute object of an object of type "EOObject" (meaning, the actual type cannot be inferred), the Java Reflection API is used to instantiate it dynamically.

All objects in the EO runtime environments have the same Java type "EOObject". From the Java Virtual Machine perspective, all of the objects have the same type with identical set of fields, methods and constructors. Hence, no compile-time type checks are performed. As showed above, most verifications are done at runtime through Java Reflection API dynamically. Thus, the proposed model has a dynamic type system.

B. The Implementation of the Model

C. Limitations and Problems of the Model

V. Approbation of the Proposed Model

This section provides the report on the conducted approbation experiments aiming to measure and compare the performance of the existing and proposed models.

A. Experiment Design

The approbation experiment was designed as follows. The output target Java sources produced by the implementations of the existing and the proposed transcompilation models for several algorithms implemented in EO (specifically, recursive factorial, array merge sorting) were benchmarked by means of the "Java Microbenchmark Harness" (JMH) micro-benchmarking utility provided by Oracle [6]. The benchmarked quantity evaluated in the experiment was time of execution of one algorithm run for each model. All the algorithms used in the experiment relied on recursive, so both models used stack hugely. Hence, JMH was allocated with 64 megabytes of stack for each tested model. Such an amount of memory was used to benchmark as complex tasks as possible.

B. Performance Comparison of the Models

Table I shows the results of the experiment.

TABLE I. Comparison of execution times

Algorithm	Table Column Head		
	n	***Existing, ms***	***Proposed, ms***
Recursive factorial of n	1	1.41	0.09
	10	14.71	0.14
	100	1244.75	0.54
	1000	124462.43	3.62
	10000	—	36.56
	100000	—	442.22
Merge sort of an array of length n	1	0.92	0.24
	2	21.48	7.32
	3	1257.56	91.69
	4	1857.67	262.47
	5	432938.08	3386.93
	6	—	8764.95
	7	—	15167.25
	8	—	35243.34

VI. Discussion

VII. Conclusion

To conclude, this work proposes a transcompilation model of EO programs to Java source code, offers a renewed EO transpiler and the standard object library implementing the proposed model, compares the proposed and the existing models, and benchmarks their performance on several algorithms to show the achieved enhancement. The findings of this work is to facilitate the EO language to evolve from an undeveloped state and be prepared for further assessments on practical applicability, interoperability with Java, performance in enterprise applications, and other analyses and improvements.

References

[1] Y. Bugayenko, "Reduced Calculus for Object Oriented Programs." [Accessed 28 April 2021].

[2] E. Bugaenko, Elegant Objects, 1st ed. Palo Alto, CA, USA: CreateSpace Independent Publishing Platform, 2017.

[3] Y. Bugayenko, "Elegant Objects", Elegant Objects. [Online]. Available: https://www.elegantobjects.org/. [Accessed: 01- Apr- 2021].

[4] "cqfn/eo", GitHub. [Online]. Available: https://github.com/cqfn/eo. [Accessed: 01- Apr- 2021].

[5] A. Church, The calculi of lambda-conversion. Princeton, NJ: Princeton Univ. Press, 1985.

[6] "openjdk/jmh", GitHub. [Online]. Available: https://github.com/openjdk/jmh. [Accessed: 28- Apr- 2021].

[7] S. Thompson, Type theory and functional programming. Wokingham et al.: Addison-Wesley, 1991.

[8] S. Peyton Jones, The implementation of functional programming languages. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides and G. Booch, Design Patterns: Elements of Reusable Object-Oriented Software, 1st ed. Addison-Wesley Professional, 1998.

[10] "Nested Classes (The Java™ Tutorials > Learning the Java Language > Classes and Objects)", Docs.oracle.com, 2021. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html. [Accessed: 03- May- 2021].

[11] "Local Classes (The Java™ Tutorials > Learning the Java Language > Classes and Objects)", Docs.oracle.com, 2021. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html. [Accessed: 03- May- 2021].