ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 4, April 2014

On the Design of an Object Oriented Programming Language

Mokua Richard

MSc, Dept. of Computing, ICSIT, JKUAT, Kenya

ABSTRACT: This paper presents work done to address two issues of interest to both Programming Language (PL) theory and software development: 1) The inadequacies of mainstream Object Oriented Programming Languages used in the software industry such as Java, C# and C++ and 2) The design and implementation of a statically typed Object Oriented Programming Language that addresses some of the issues identified above.
Research was conducted through critical analysis of existing Object Oriented Programming Languages (OOPL) as well as a literature review of journal and conference publications in that area. The aim was to elicit evidence of PL constructs that had been found through previous experience to lead to poor Software Engineering practices such as increased amount of bugs, poor maintainability, late (i.e. runtime) detection of errors, poor usability and low programmer productivity.
This work has produced key benefits that include a deeper understanding of PLs specifically OOPLs, and an improved comprehension and appreciation of the nuances of PL design.
The findings have the potential to benefit PL researchers and designers in various ways.
We consider that the contributions of this work are that a list of the language constructs (e.g., Static Variables, Lack of Object Level Encapsulation, Presence of Primitive Types) that seem to lead to poor Software Engineering practices with current OOPL have been identified. A further significant contribution is the production of a new OOPL designed to act as proof of concept to illustrate how these issues can be addressed.

KEYWORDS: Object Oriented Programming Language; Compilers; Software Engineering; Type Systems; Compiler Design and Construction

I. RELATED WORK

The concept of traits were first introduced by Smalltalk, but the version used here is based on [28]. The syntax of the designed language closely follows that of Scala [22]. Object based encapsulation is an extension of the one implemented in Scala [22] and also an adaptation of the one in Newspeak. Elimination of static state is partly an adaptation of work by Bracha [3] in which he does the same for his *dynamically typed language*. In this paper, the same concept is applied to *statically typed language*. Uniform object model was inspired by Smalltalk [15], Newspeak[3] and Scala [22]. In implementing the compiler, a lot was learnt from browsing Java compiler source code from the OpenJDK project and the open source Fortress project.

II. INTRODUCTION

Most programming languages can be classified into families based on their model of computation [24]. Declarative languages focus on instructing the computer *what* to do while imperative languages focuses on *how* the computer should do it.

Declarative languages can further be divided into the following sub-categories:

- *Functional languages* employ a computational model based on the recursive definition of functions. They take their inspiration from the lambda calculus [5]. In essence, a program is considered a function from inputs to outputs, defined in terms of simpler functions through a process of refinement. Languages in this category include Lisp [18], ML [20] and Haskell [23].
- *Dataflow languages* model computation as the flow of information (tokens) among primitive functional nodes. Val (Ackerman and Jack, 1979) is an example of a language from this category.
- *Logic* or *constraint-based languages* take their inspiration from predicate logic. They model computation as an attempt to find values that satisfy certain specified relationships, using a goal-directed search through a list of logical rules. Prolog [12] is the best-known logic language.

Imperative languages are divided into the following subcategories:

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 4, April 2014

- *von Neumann languages* are the most familiar and commonly used programming languages. They include FORTRAN [13], Ada 83[6], C [19], and all of the others in which the basic means of computation is the modification of variables [20].
- *Object-oriented languages* are comparatively recent, though their roots can be traced to Simula 67[14] .Most are closely related to the von Neumann languages but have a much more structured and distributed model of both memory and computation. Rather than picture computation as the operation of a monolithic processor on a monolithic memory, object-oriented languages picture it as interactions among semi-independent objects, each of which has both its own internal state and subroutines to manage that state. Smalltalk [15] is the purest of the object-oriented languages; C++ [25] and Java [16] are the most widely used.

Programming Languages can also be categorized based on if they have a type system or not. In *typed languages*, program variables have an upper bound on the range of values that they can assume. On the other hand, *un-typed languages* do not restrict the range of variables [11].

III. CONTRIBUTIONS

The contributions of this paper are as follows: Identify language constructs whose use in writing programs violate the Software Engineering principles of Testability, Reusability, Security and Expandability. Demonstrate through case studies and literature review ways in which such constructs affect the design principles identified above. Design the syntax and semantics of a language that solves the issues identified. We then proceed to give a detailed overview of both the syntax and semantics of the new language. Develop a prototype compiler for the languages. We give an overview of the design of the compiler and the challenges experienced and tradeoffs made in the design and implementation.

IV. CRITIQUE OF THE STATE OF THE ART

This research was undertaken using two key methods; first is the analysis of programming language theory and second experimenting with open source programming languages. We were able to identify several language constructs in most modern widely used Programming Languages that if used in the development of a software system they could lead to violation of some of the software engineering principles specified in the ISO/IEC 9126-1:2001 Standard.

a) Presence of Static Variables

If a field is declared static, there exist exactly one incarnation of the field, no matter how many instances (possibly zero) of the class may eventually be created. A static field, sometimes called a class variable, is incarnated when the class is initialized [16].In Java, static means one per class, not one for each object no matter how many instance of a class might exist. This means that a static variable can be used without creating an instance of the class. Static variable present a number of challenges in the language:

- *Static variables increase cases of security vulnerabilities*

This is due to two main factors , there is no way to check whether the code that changes such variables has the appropriate permissions and any mutable static state can cause unintended interactions between supposedly independent subsystems.

- *Static variables leads to systems that are not re-entrant.*

It is not possible to have several concurrent executions of the software in the same VM. In the paper [21], the authors describes a number of disadvantages that they encountered due to use of static variables in the first version of the Scala [22] compiler. Since all references between classes were hard links, the author could not treat compiler classes as components that can be combined with different other components. This, in effect, prevented piecewise extensions or adaptations of the compiler. Another issue is that since the compiler worked with mutable static data structures, it was not re-entrant, i.e. it was not possible to have several concurrent executions of the compiler in a single VM. This made it a problem for using the Scala compiler in integrated development environment such as Eclipse.

- *Static variables increase the startup time*

They encourage excess initialization up front. The Java Virtual Machine Specification [26] specifies that the static initializers and class variable initializers are executed in textual order. They may not refer to class variables declared in the class whose declarations appear textually after the use, even though these class variables are in scope. This restriction is designed to catch, at compile time, most circular or otherwise malformed initializations. Not to mention the complexities that static initialization engenders: it can deadlock,

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 4, April 2014

applications can see uninitialized state, and it is hard to compile efficiently (because there is need to test if things are initialized on every use).

• *Static variables are bad for distribution.*

Static state needs to either be replicated and synced across all nodes of a distributed system, or kept on a central node accessible by all others, or some compromise between the former and the latter. This is all difficult, expensive and unreliable.

• *Static variables make it difficult to do testing of code*

The reason is that states in static values may be kept between unit tests because the class or *dll* is not unloaded and reloaded between each unit test. This violates the principle that unit tests should be independent of each other, and can result in tests passing and failing depending on the order in which they are run.

b) Lack of Object Level Encapsulation

Most mainstream Object Oriented languages use *class based encapsulation.* The idea is that privacy is *per class*, not *per object*. This makes it possible to violate data abstraction as shown below.

```
class C {
private i :String ;
def public m1(v : C) : Unit= { v.i = "XXX"}
}
```

As the above code illustrates, class based encapsulation does not protect one object from another since one object is able to access (and modify) the **private** attributes of another object. An alternative to class-based encapsulation is *object based encapsulation*. Privacy is per object. A member **M** marked with **private** modifier can be accessed only from within the object in which it is defined. That is, a selection **p.M** is only legal if the prefix is **this** or **O.this**, for some class **O** enclosing the reference.

```
class C {
private i : String ;
def public m1(v : C) : Unit= { v.i = "XXX"} //Error
def public m2():Unit = { this.i = "YY";} //Ok
}
```

A member marked private is visible only inside the object that contains the member definition.

c) Method Lookup Strategy

In general, the semantics of a method invocation that has no explicit target (receiver) are that method lookup begins by searching the inheritance hierarchy of *self* (*this*); if no method is found, then the lookup procedure is repeated recursively at the next enclosing lexical level. This notion is described in detail in the Java Language Specification [16] in section 15.12 (Method Invocation Expressions). Situations like the following can arise:

```
class Sup { }
class Outer {
        int m(){ return 91}
        class Inner extends Sup {
                int foo(){return m()}// case 1: new Outer.Inner().foo() = 91
                }
        }
}
```

The expectation is that a call to **foo** will yield the result 91, because it returns the result of the call to **m**, which is defined in the enclosing scope [3]. Consider what happens if one now modifies the definition of Sup:

```
class Sup {
        int m(){ return 42}// case 2: new Outer.Inner().foo() = 42
}
```

The result of calling **foo** is now 42. This is undesirable; since the behaviour of the subclass changes in a way that its designer cannot anticipate. The classic semantics whereby inherited members may obscure lexically visible ones are counterintuitive. Lexically visible definitions should be given priority over inherited ones, either

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 4, April 2014

implicitly via scope rules or by requiring references to inherited features to explicitly specify their intended receiver. Retain implicit receivers for both self sends and outer sends, but reverse the priority so as to favour sends to names with locally visible definitions.

d) Presence of Primitive Types

Most mainstream statically typed object oriented languages divide their types into two categories, primitive (built in) types e.g. **int, char, long, double, float, short** and reference types e.g. **Integer, String**
This dichotomy presents a number of problems:

1) *Dichotomy of basic semantics.* Features of the language carry different meaning depending on the type of entity being dealt with. For example the built in operator == means different things depending on whether the variable types are primitives or reference types [16] .
2) *Primitives cannot be used where objects are expected.* For example, in the Java Language the container **Vector** cannot be used to store variables of primitive types, since it's designed to store variables of reference type **Object**.
3) *Primitive types advertise their representation to the world*. As one example, consider type **char**. When Java was introduced, the Unicode standard [27] required 16 bits. This later changed, as 16 bits were inadequate to describe the world's characters. In the meantime, Java had committed to a 16 bit character type. Now, if characters were objects, their representation would be encapsulated, and nobody would be very much affected by how many bits are needed.
4) *Primitive types necessitate the existence of special code which leads to the undoing of polymorphism*. This is due to the fact that we cannot send messages to variables of primitive type. For example the **String** class has a **static** method **valueOf** that produces a **String** representation of its argument. For reference arguments, the Objects **toString** method is invoked.
5) The inclusion of primitive types forces Java Reflection API to be inconsistent and essentially broken to accommodate them.

V. CASE STUDIES

A. *Static Variables : Scala Compiler:*

This section relates the experience of the Scala Team in the implementation of two different versions of the Scala compiler as described in the paper Scalable Component Abstractions [21]. The Scala compiler consists of several phases. All phases after syntax analysis work with the symbol table module. The table consists of a number of modules including: ***Names*** module that represents symbol names. A name is represented as an object consisting of an index and a length, where the index refers to a global array in which all characters of all names are stored. ***Symbols*** modules that represent symbols corresponding to definitions of entities like classes, methods, variables in Scala and Java modules. A module ***Types*** that represents types. A module ***Definitions*** that contains globally visible symbols for definitions that have a special significance for the Scala compiler.
In previously released versions of the Scala compiler, all modules described above were implemented as top-level classes using Java language, which contain static members and data. For instance, the contents of names were stored in a static array in the Names class. This technique has the advantage that it supports complex recursive references. But it also has two disadvantages, since all references between classes were hard links, the compiler classes could not be treated as components that can be combined with different other components. This prevented piecewise extensions or adaptations of the compiler and second, since the compiler worked with mutable static data structures, it was not **re-entrant**, i.e. it was not possible to have several concurrent executions of the compiler in a single JVM. This was a problem for using the Scala compiler in an integrated development environment such as Eclipse.
The Scala Team solved the above problem introduced by static references through the use of nested classes and doing away with static references. In that way, they arrived at a compiler without static definitions. The compiler is by design re-entrant, and can be instantiated like any other class as often as desired.

B. *Method Lookup Strategy : Newspeak Programming Language:*

In the paper [3], Gilad Bracha provides his experience on the implementation of method lookup mechanism for the Programming Language ***Newspeak***. Newspeak is a dynamically typed class based language which is a descendant of Smalltalk. The paper presents alternative interpretations of the semantics of method lookup: Require all sends to

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 4, April 2014

have an explicit receiver as in Smalltalk. The problem with this solution is that it's overly verbose. Require outer sends to have an explicit receiver. Require all self sends to have an explicit receiver. Given that outer sends have an implicit receiver, it makes no sense to treat locally defined self sends differently, so we interpret this as only requiring all inherited self sends to have an explicit receiver. Retain implicit receivers for both self sends and outer sends, but reverse the priority so as to favor sends to names with locally visible definitions.

C. *Uniform object Model:*

In OOPL there has always been distinction between "primitive" or "built-in" and user defined types. The paper [7], shows how an object-oriented language can be defined without any primitive types at all, and yet achieve the same run-time efficiency as languages that make use of primitive types (at the expense of greater compile-time effort). The authors' quote the following as advantages of having a uniform object model in a language: The programming model is simplified because the distinction between primitives and objects has been removed; and the language design is simplified and more easily verifiable because a larger amount of the language is in libraries, and there is no need for large numbers of rules for primitive types that must be included in the language specification and verified on an ad-hoc basis.

The paper [22] describes how Scala uses a pure object-oriented model. Every value is an object and every operation is a message send. Every class in Scala inherits from class *Scala.Any*. Subclasses of *Any* fall into two categories: the value classes which inherit from *scala.AnyVal* and the reference classes which inherit from *scala.AnyRef*. Another aspect of Scala's unified object model is that every operation is a message send, that is, the invocation of a method. For instance the addition **x + y** is interpreted as **x. +(y)** i.e. the invocation of the method **+** with **x** as the receiver object and **y** as the method argument.

VI. LANGUAGE SYNTAX AND SEMANTICS

The Lexical Structure of the language closely resembles that of Java; in particular, lines are terminated by the ASCII characters CR, or LF, or CR LF. White space is defined as the ASCII space, horizontal tab, and form feed characters, as well as line terminators. Both single-line comments and multi-line comments are supported.

The language has a nominal type system [31] with some elements of structural typing. In particular, it has the following kind of types , Class Type is introduced through a Class declaration. The name of the Class is the type. A class type **e** is a subtype of every type that appears on its **extends** clause. The Trait Type is introduced through a Trait declaration. The name of the Trait is the type. A trait type is a subtype of every type that appears on its **extends** clause. The Function type is introduced through block closure declaration.

A Compilation Unit consists of a package declaration, followed by a sequence of type definitions. Class declaration contains the name of the class, a *modifier* that restricts the visibility of the class constructor, a list of formal value parameters for the default constructor and an optional list of well-formed trait names that are accessible from this class declaration. The class body defines the class members i.e. fields, methods, constructors and nested class definitions.

A Trait declaration contains the trait name followed by an optional type parameter clause and an extend clause finally followed by the trait body. When a trait extends others, it means that it inherits the methods from those traits, and that the type defined by that trait is a subtype of the types of traits it extends.

We give an overview of the expressions in the language:

1) *Instance Creation Expression*: Instance Creation Expression has the form new c where c is a constructor invocation. Let *T* denote the type of *c*, then *T* must denote a non-abstract subclass of *Object.*
2) *This and Super*

this refers to the object reference of the inner most class or trait enclosing the expression. The type of ***this*** is the type of the class or trait. A reference *super.m* refers statically to a member *m* in the super-type of the innermost class or trait containing the reference. It evaluates to a member *m'* that has the same name as *m.*

3) *Method Invocation*:

A Method Invocation expression has the form $e.m(e_0,...,e_n)$.The expression *e*, if present, must evaluate to an object expression. Let the expression *e* have the type *T* given by the definition *D.* Then:

a. *D* must be either a Trait or a Class.
b. Further, *D* must define a method of the form $m(e:T_0,...,e_l: T_l)$. where:

- *l* must be equal to *n*

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 4, April 2014

- for every *i* in *0...n*, the type of the parameter value expression ,*e* in $e.m(...,e_i,...)$ must conform to the expected type of the corresponding formal parameter declaration $m(...,e_i:T_i,...)$.If the method invocation expression has the form $m(e_0,...,e_n)$.

Then search for *m* in the following order:

- Search for a field named *m* in the enclosing class declaration.
- If *m* is a field, then its type must be function type and the expression is a *Lambda Invocation* .
- Else, search for a method named *m* recursively in the outer class enclosing this class declaration if any.
- If not found, search the Traits that this class extends for a method named *m.*
- If still not found, return the error *method not found.*

4) *Blocks*

A Block expression has the form $\{s_0,... s_n, e_0\}$. The result of evaluating the Block expression is the value of the evaluation of the last expression in the block. The type of the last expression in the block must conform to the type of the Block expression. Let the expected type of the Block expression $e = \{s_0,...,s_n,e_0;\}$ be *T*, then the type of the expression e_0 must conform to *T.*

5) *Assignments*

An Assignment Expression has the form $x = e$ .The assignment changes the current value of *x* to be the result of evaluating the expression *e*. The type of *e* is expected to conform to the type of *x*.

6) *If Expressions*:

An if expression has the form: ***if*** (e_0) e_1 ***else*** e_2.The expression e_0 must conform to a **Boolean** type. The type of the expression e_1 and e_2 must conform to the expected type of the *if* expression. The expression to be executed is chosen based on the results of the evaluation of the *Boolean* expression e_0

7) *While Loop Expression*

A While loop has the form ***while***$(e_0)\{$ $e_1\}$.The expression e_1 is repeatedly evaluated until the evaluation of the expression e_0 results in a false value. If e_0 evaluates to false, then the expression e_1 is not evaluated. The type of the expression e_0 must conform to a *Boolean* type. The type of the e_1 expression must conform to the type of the while expression.

8) Do Loop Expressions

A Do Loop has the form $do\{e_1\}while(e_0)$.

The expression e_1 is evaluated; if e_0 evaluates to false, the expression e_1 is not evaluated. Otherwise the expression e_1 is repeatedly evaluated until the evaluation of the expression e_0 results in a false value. The type of the expression e_0 must conform to a *Boolean* type. The type of the e_1 expression must conform to the expected type of the do loop expression.

9) *Lambda Expression*

Lambda Expression has the form $\#(p_0: T_0,...,p_1: T_n) \Rightarrow e$.

The formal parameters $\#(p_0: T_0,...,p_1: T_n)$ must be pair-wise distinct. The scope of the parameters is the expression *e*. The expression must conform to the expected type of the Lambda expression.

VII. COMPILER

The compiler is written in Java. It compiles programs and generates JVM byte-codes [26] which can execute on the JVM. The compilation is done over a number of phases. These phases include: Lexical Analysis in which the source program is transformed to a stream of tokens: symbols such as identifiers, literals, operators, keywords and punctuation. Comments and blank spaces are discarded. The parser constructs the Abstract Syntax Tree(AST) from the token stream during parsing. Semantic analysis involves several stages.

Name Analysis: When defining a name if the name is already in the local environment: the identifier is already declared. Else, the new name is inserted in the environment. When looking up a name, first look in the local environment. If it is found we are done, otherwise repeat in the next environment on the search path. If there are no more environments the identifier is not declared.

The Type-checker performs the following tasks, determine the types of all expressions and checking that values and variables are used consistently with their definitions and with the language semantics.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 4, April 2014

Reachability Analysis: This phase involves carrying out a conservative ***flow analysis*** to make sure all statements are reachable. There must be some possible execution path from the beginning of the constructor, method or instance initializer that contains the expression to the expression itself.

Definite (un)assignment analysis: consists of two parts, in the first part, each local variable and every blank *val* must have a definitely assigned value before any access of its value occurs and every blank *val* variable must be assigned at most once; it must be definitely unassigned when an assignment to it occurs.

Closure conversion transforms a program in which functions can be nested and have free variables into an equivalent one containing only top level functions.

Algorithm

i. The closing of functions through the introduction of environments. Functions are closed by adding a parameter representing the environment, and using it in the function's body to access free variables. Function abstraction must create and initialize the closure and its environment; Function application must extract the environment and pass it as an additional parameter.

ii. The hoisting of nested, closed functions to the top level. Once they are closed, nested anonymous functions are hoisted to the top level and given an arbitrary name. The original occurrence of the nested function is replaced by that name.

The input to the byte-code generation is an attributed AST. The algorithm traverses this AST generating byte-code for each of the constructs found in the tree. The algorithm uses object web byte code generation library.

VIII. CONCLUSION AND FUTURE WORK

We have presented several programming language constructs that we believe when used in large software projects they can lead to software that is of poor quality. We then demonstrated by use of four case studies the problems caused by some of the constructs based on real life large software projects. Each of this construct is avoided in some of the existing Statically Typed Object Oriented Programming Language, but we believe we are the first to eliminate all of them in one STOOPL language.

There are several issues that need to be addressed through further research: A formalization of the design of the language along with proofs of type safety and the implementation of a production quality compiler.

REFERENCES.

1. Ackerman, W.B.and Jack B. D. ,VAL: a Value oriented Algorithmic Language, 1979.
2. Ancona, D., Lagorio, G. & Zucca, E., Jam - A Smooth Extension of Java with Mixins, ECOOP00 European Conference on Object Oriented Programming. Springer, pp. 154-178, 2000.
3. Bracha, G, On the Interaction of Method Lookup and Scope with Inheritance and Nesting, 2010.
4. Bracha, G. & Cook, W., Mixin-Based Inheritance, ACM Sigplan Notices. ACM, pp. 303-311, 1990.
5. Barendregt, H.P., The Lambda Calculus its Syntax and Semantics, North-Holland, 1981.
6. Barnes, J., Ada 2005 Rationale, Springer, 2008.
7. Bacon, D.F., Kava: a Java dialect with a uniform object model for lightweight classes, Concurrency: Pract. Exper.,2003.
8. Boehm, B.W., Brown, J.R. & Lipow, M., Quantitative evaluation of software quality, 2008.
9. Cielecki, M., Fulara, J. & Jakubczyk, K., Propagation of JML non-null annotations in Java programs of programming in Java, 2006.
10. Cardelli, L., Type systems. ACM Computing Surveys, 28(1), pp.263-264, 1996.
11. Clocksin, W.F. & Mellish, C.S., Programming in Prolog, Springer-Verlag, 1981.
12. Chivers,I and Sleightholme,J., Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003 and 77. Springer: London, U.K. Springer, 2006.
13. Dahl, O.-J., Myrhaug, B. & Nygaard, K.,SIMULA 67. Common Base Language, 1968.
14. Goldberg, A, and David R.(1985).Smalltalk-80: the Language and Its Implementation. Reading, MA: Addison-Wesley.
15. Gosling, J. et al., The Java Language Specification, Third Edition, Addison Wesley, 2005.
16. Halloway, S, Programming Clojure S. Davidson Pfalzer, ed., Pragmatic Bookshelf, 2009.
17. Harrison, M,The Programming Language LISP: Its Operation and Applications, MIT Press,1967.
18. Kernighan, B.W. & Ritchie, D.M, The C Programming Language, Prentice Hall, 1978.
19. Milner, R., Tofte, M. & Harper, R., The Definition of Standard ML, MIT Press, 1990.
20. Odersky, M. & Zenger, M.,Scalable component abstractions. ACM SIGPLAN Notices, 40(10), p.41, 2005.
21. Odersky, M. et al.,An Overview of the Scala Programming Language, Second Edition, 2006.
22. O'Sullivan, B., Goerzen, J. & Stewart, D., Real World Haskell, O'Reilly Media, 2008.
23. Scott, M.L. , Programming language pragmatics, Morgan Kaufmann Pub, 2000.
24. Stroustrup, B., The C++ Programming Language, Addison-Wesley,1997.
25. Lindholm, T. & Yellin, F, The Java Virtual Machine Specification, Addison-Wesley,1999.
26. Rossum, G.V. & Drake, F.L., Unicode HOWTO. History, 172(7), p.10, 2010.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 4, April 2014

27. Scharli, N. et al. ,Traits: Composable units of Behaviour, ECOOP 2003–Object-Oriented Programming, p.327–339, 2003.
28. Schärli, N. et al ,Traits: Composable Units of Behavior. Technical Report, 2743, pp.248-274, November 2002.
29. Nutter, C. et al., Using JRuby, Pragmatic Programmers, 2010.
30. Pierce, B.C., Types and Programming Languages, The MIT Press, 2002.
31. Taivalsaari, A., On the notion of inheritance. ACM Computing Surveys, 28(3), pp.438-479, 1996.
32. Borning, A.H. & Ingalls, D.H.H., Multiple Inheritance in Smalltalk-80. In Proceedings at the National Conference on AI. pp. 234-237, 1982.
33. Keene, S.E., Object-Oriented Programming in Common-Lisp, Addison Wesley, 1989.
34. Meyer, B., Object-Oriented Software Construction, Second Edition, Prentice Hall PTR, 1997.
35. Schaffert, C. et al., An Introduction to Trellis/Owl, ACM Sigplan Notices. pp. 9-16, 1986.
36. Flatt, M., Krishnamurthi, S. & Felleisen, M., Classes and Mixins, Conference Record of POPL98 The 25th ACM SIGPLANSIGACT Symposium on Principles of Programming Languages, ACM Press, pp. 171-183, 1998.
37. Mens, T. & Van Limberghen, M., Encapsulation and Composition as Orthogonal Operators on Mixins: A Solution to Multiple Inheritance Problems , Object Oriented Systems, 3(1), pp.1-30, 1996.
38. Moon, D.A., Object-Oriented Programming with Flavors, Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications OOPSLA. ACM Press, pp. 1-8.
39. David A. Moon. Object-oriented programming with flavors. In Proceedings OOPSLA 86, ACM SIGPLAN Notices, volume 21, pages 18,November 1986.

BIOGRAPHY

Mokua Ombati Richard is a MSc. Software Engineering student in the Computing Department, Jomo Kenyatta University Of Agriculture And Technology. His research interests are Programming Languages, Compilers, and Computer Security.