AND IMPLEMENTATION OF AN OBJECT ORIENTED PROGRAMING LANGUAGE

R. Mokua[1], W. Mwangi[2] and J. M. Kanyaru[3]
[1,2]Institute of Computer Science and Information Technology, Jomo Kenyatta University of Agriculture and Technology, Nairobi, Kenya
[3]Software Systems Research Center, Bournemouth University, United Kingdom
E-mail: mokua.richard@gmail.com

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. 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.

Key words: Object oriented programming language, compilers, software engineering, type systems, compiler design and construction

1.0 Introduction

Most programming languages can be classified into families based on their model of computation (Scott,2000). 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 (Barendregt, 1981). 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 (Harrison, 1967), ML (Milner et al, 1990) and Haskell (O'Sullivan et al, 2008).

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 (Clocksin and Mellish, 1981) is the best-known logic language.

Imperative languages are divided into the following subcategories:
von Neumann languages are the most familiar and commonly used programming languages. They include FORTRAN (Chivers and Sleighthome, 2005), Ada 83 (Barnes, 2008), C (Kernighan and Dennis, 1978), and all of the others in which the basic means of computation is the modification of variables (Scott,2000)

Object-oriented languages are comparatively recent, though their roots can be traced to Simula 67(Dahl et al, 1968) .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 (Goldberg, A, and David R , 1985) is the purest of the object-oriented languages; C++ (Stroustrup , 1997) and Java (Gosling et al , 2005) 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 (Cardelli, 1996).

In Typed languages, the compiler can enforce good behavior by performing static (i.e., compile time) checks to prevent unsafe and ill behaved programs from ever running. These languages are said to be statically checked (Cardelli, 1996).On the other hand, the type checking may be delayed until runtime, in which case the language is said to be dynamically typed (Cardelli, 1996).

Examples of Object Oriented Languages that are statically typed include Java and C++ while those that are dynamically typed include Clojure (Halloway, 2009) and Smalltalk.
According to research done by the TIOBE Index , Object-oriented statically typed languages have been the most popular category of Programming Languages for more than 4 years now. Due to the fact that these are the most popular languages, it makes sense to invest time and energy in improving such languages so that we can improve the programmer's productivity.

2.0 Contributions

The ISO/IEC 9126-1:2001 Standard identifies a set of non-functional requirements which enhances the quality of the software program. Some of these requirements are:
(I) *Maintainability* - Effort required to locate and fix an error in a program.
(II) *Testability* - Effort required for testing the programs for their functionality.
(III) *Portability* - Effort required for running the program from one platform to other or to different hardware.

(IV) *Reusability* - Extent to which the program or its parts can be used as building blocks for other programs.
(V) *Interoperability* - Effort required to couple one system to another.
(VI) *Security* - The capability of the software product to protect information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them.
(VII) *Expandability* - The ease with which the software can be modified to add functionality.
(VIII) *Simplicity* - Degree to which a program is understandable without much difficulty.
(IX) *Integrity* - Degree to which a program can continue to perform despite some violation of the assumptions in its specification. This implies, for example, that the program will properly handle inputs out of range, or in different format or type than defined, without degrading its performance of functions not dependent on the non-standard inputs (Boehm *et al*., 2006).

It's important that a programming language be able to provide constructs that enable the programmer to develop systems that observe the above software engineering principals. Further, the language should make it difficult for the programmer to implement systems that violate the above principles as much as possible.

Historically, Statically Typed Object Oriented Programming Languages (STOOPL) have included some features that have been known to lead to software systems that violate the above software engineering principles.

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; In Section II, we give a detailed list describing the issues identified; demonstrate through case studies and literature review ways in which such constructs affect the design principles identified above. In Section III of the paper, we delve in details into several case studies illustrating the effect of some of the identified constructs in real life large open source software projects and then how the issues were addressed; design the syntax and semantics of a language that solves the identified issues. We then continue to give a detailed overview of both the syntax and semantics (Section IV) of the new language and develop a prototype compiler for the languages. In Section V, we give an overview of the design of the compiler and the challenges experienced and tradeoffs made in the design and implementation.

2.1 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.

2.2 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 (Gosling et al , 2005).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:

2.3 Static variables increase cases of security vulnerabilities

This is due to two 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.

The key idea is that static state represents an ambient capability to do things to a system that may be taken advantage of by unauthorized persons or systems. Static variables leads to systems that are not re-entrant, i.e., it's not possible to have several concurrent executions of the software in the same VM. In the paper (Odersky and Zenger , 2005), the authors describe the following disadvantages that they encountered due to use of static variables in the first version of the Scala (Odersky *et al*., 2008) compiler.

Since all references between classes were hard links, they 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 and 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 was a problem for using the Scala compiler in integrated development environment such as Eclipse.

2.4 Static Variables Complicate Memory Management

According to the Java Language Specification (Gosling et al , 2005) section 12.7, Class unloading is an optimization that helps reduce memory use. However, if a class or interface ***C*** was unloaded while its defining loader was potentially reachable, then might be reloaded. This reloading needs to be transparent. However, reloading may not be transparent (and hence the class cannot be unloaded) if the class has; static variables (whose state would be lost); static initializers (which may have side effects) and native methods (which may retain static state).

2.5 Static Variables Increase the Startup Time

They encourage excess initialization up front. The Java Virtual Machine Specification (Lindholm and Yellin, 1999) 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, applications can see uninitialized state, and its hard to compile efficiently (because there is need to test if things are initialized on every use).

Use of static variables can easily lead to concurrency problems e.g. deadlocks; 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 and static variables encourages designs using global variables hence makes it harder to re-use code.

3.0 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 labeled private is visible only inside the object that contains the member definition.

4.0 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 (Gosling et al, 2005) 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 (Bracha, 2010). 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 behavior 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 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 favor sends to names with locally visible definitions.

5.0 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:

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 (Gosling *et al.,* 2005) .

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.

Primitive types advertise their representation to the world

As one example, consider type char. When Java was introduced, the Unicode standard (Rossum and Drake, 2010) 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.

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.

```
public static String valueOf(Object o){
        return o == null? null :o.toString();
}
```

But this breaks down for primitive types necessitating overloading the valueOf() method for each of the primitive type.

```
public static String valueOf(boolean o){
        return o ? true :false;
}
```

1) The inclusion of primitive types forces Java Reflection API to be inconsistent and essentially broken to accommodate them.

```
String s =" test";
s.getClass(); // this is ok
int i = 20;
i.getClass();// compile error
```

6.0 Existence of Null Pointers

In current imperative languages, references (or pointers) can hold a special value meaning reference to nothing. This value is called *null* in Java. The **null** type has one value, the **null** reference, represented by the literal **null** (Gosling et al, 2005).Dereferencing a null at runtime results to a runtime error (*NullPointerException*) causing the current thread to exit if the exception is not handled.

Disadvantages of Null Pointers include, makes Java programs unsafe because when it occurs at runtime it causes system to crash; exceptions due to dereferencing a null pointer are the most common type of error in Java programs (Hovemeyer and Spacco , 2006;Cielecki et al ,2006) and It's among the Top 10 web application security risks.

7.0 Improving Opportunities for Reuse

Inheritance is commonly regarded as one of the fundamental features of object-oriented programming (Taivalsaaril, 1996). Over the years, researchers have developed various inheritance models (Borning and Ingalls ,1982; Keene, 1989 ; Meyer, 1997) and *mixin* inheritance (Schaffert et al,1986;Ancona et al, 2000; Bracha and Cook, 1990;Flatt and Felleisen, 1998;Mens and van Limberghen,1996).All of these models have their conceptual and practical shortcomings with respect to reusability (Scharli et al, 2003). Inheritance also has other problems such as implementation difficulties (Cardelli, 1997; Taivalsaari, 1996) and conflicts between inheritance and *sub-typing*.

Single Inheritance - A class can inherit from at most one *superclass*. It is not expressive enough to allow the programmer to factor out all the common features shared by classes in a complex hierarchy. Hence single inheritance sometimes forces code duplication (Scharli et al, 2003).

Multiple Inheritance - enables a class to inherit features from more than one parent class, thus providing the benefits of better code reuse and more flexible modeling. However, multiple inheritance uses the notion of a class in two competing roles: the generator of instances and the unit of code reuse. This gives rise to the following difficulties (Scharli *et al.*, 2003), conflicting features; accessing overridden features and factoring out generic wrappers.

*Mixin Inheritance,*Total ordering,dispersal of glue code; ragile hierarchies.

Traits (Scharli *et al.*, 2003; Scharli *et al.*, 2002) improve code-sharing in Smalltalk by providing a means to reuse such behavior. Traits are a mechanism for code reuse that complements single inheritance. Traits, like classes, are containers for methods. But, unlike classes, traits have no fields. Traits, like abstract classes, cannot be instantiated directly; instead, they are composed into classes (which are instantiable).

Presence of Type Based Overloading

Constructor and method overloading is a type of polymorphism where different functions with the same name are invoked based on the data types of the parameters passed.

```
class VehicleUtilities {
        int numberOfAxles(Vehicle v) { return 2;}
        int numberOfAxles (Truck t){ return 3;}
}
```

In general, overloading means that a function name or an operator has two or more distinct meanings. When you use it, the types of its operands are used by the language to determine which meaning should apply. In Java, the programmer can declare several different functions, in the same scope, with the same name, but different parameter types. A call using this function name is resolved at compile time to one of the several functions by looking at the types of the actual parameters (Gosling *et al*., 2005).

Problems with type based method overloading include:
Risk of ambiguity
In most overloading schemes, you can create situations where you cannot decide which method to call and therefore declare the call illegal. Unfortunately, as the type hierarchy evolves, legal code can become illegal, or simply change its meaning. This means that existing code breaks when you recompile, or does the wrong thing if you do not.
Overloading is open to abuse
It allows you to give different operations the same name.
Overloading makes it hard to interoperate with other languages.
It's harder to call the overloaded methods from another language. Such a language may have a different type system and/or different overload rules.
Dynamic language
JRuby (Nutter *et al*, 2010) implement multimethod dispatch to approximate the behavior of overloaded methods it needs to call. This is costly at run time, and is a burden on the language implementer.

Overloading adds complexity to the language
It tends to interact with all sorts of other features, making those features harder to learn, harder to use, and harder to implement. In particular, any change to the type system is almost certain to interact with type based overloading.

8.0 case studies

8.1 Static Variables : Scala Compiler

This section is based on 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 (Odersky and Zenger , 2005). 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. A hashmap ensures that names are unique, i.e. that equal names always are represented by the same object; 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, and a module definitions that contains globally visible symbols for definitions that have a special significance for the Scala compiler. Examples are Scala's value classes, the top and bottom classes Scala.Any and Scala.All. The structure of these modules is highly recursive. For instance, every symbol has a type, and some types also have a symbol.

In previously released versions of the Scala compiler, all modules described above were implemented as top-level classes (implemented in Java), 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. Firstly, 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. 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 VM. 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.

8.2 Method Lookup Strategy : Newspeak Programming Language

In the paper (Bracha, 2010), 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:

(i) Require all sends to have an explicit receiver as in Smalltalk. The problem with this solution is that it's overly verbose.
(ii) Require outer sends to have an explicit receiver. This also solves the problem.
(iii) 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.
(iv) 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. In Newspeak an identifier refers to the nearest lexically visible declaration, subject to overriding by subclasses. If no lexically visible binding exists, they interpret it as a self send. If one wishes to refer to an inherited method of an enclosing class, an explicit outer send expresses this intent unambiguously.

8.3 Uniform Object Model

Kava: In OOPL there has always been distinction between" primitive" or "built-in" and user defined types. The paper (Bacon, 2003), 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:

(i) The programming model is simplified because the distinction between primitives and objects has been removed; and
(ii) 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.

Scala: The paper (Odersky et al, 2006) describes how Scala uses a pure object-oriented model. Every value is an object and every operation is a message send.

(iii) **Classes** 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*.
(iv) **Operations** 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.

9.0 Language Syntax and Semantics

The language syntax and semantics is based on C, C++, C# and Java family but with a few differences, i.e., in variable declaration, the type comes after the variable name and not before e.g. *var i:int* instead of *int i*. This is useful in type inference; addition of Traits (Scharli et al, 2003) to the type system; unlike C#, C++ and Java, there is no separation of computation into *Expressions* and *Statements*; these are unified into Expression; no support for static variable and static methods; uniform object model, there is no dichotomy of primitive and non-primitive types.

9.1 Lexical Structure

The Lexical Structure of the language closely resembles that of Java; in particular

(i) *Line Terminators*: Lines are terminated by the ASCII characters CR, or LF, or CR LF.
(ii) *White Space*: White space is defined as the ASCII space, horizontal tab, and form feed characters, as well as line terminators.
(iii) *Comments*: Supports both single-line comments and multi-line comments
(iv) *Identifiers*: can be composed of a sequence of characters beginning with an alphabet and followed by alphabets or digits

(v) *Keywords*
(vi) *Literals*: There are four types of literals: Integer, Floating Point, Boolean, Character and String literals.

9.2 Type System

The language has a nominal type system (Pierce, 2006) with some elements of structural typing. In particular, it has the following kind of types:

(i) *Class types*: The 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.
(ii) *Trait types*: 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.
(iii) *Function types*: The Function type is introduced through block closure declaration.

9.3 Declarations

(i) *Compilation Unit*: A Compilation Unit consists of a package declaration, followed by a sequence of type definitions.

CompilationUnit ::= package QualId ;
[TopStatSeq]
TopStatSeq ::= TopStat {; TopStat}
TopStat ::= {Modifier} TypeDef

(ii) *Programs*: A program is a top-level class that has the method *main* of type *Array[String]Unit*. Program execution begins from the main method.

(iii) *Class Declaration*: The syntax for class declaration:

ClassDef ::= id [Modifier] ClassParamClause
[ExtendClause] ClassBody
ExtendClause ::= extends TraitTypes
TraitTypes ::= TraitType {, TraitType}
ClassParamClause ::= ([ClassParams])
ClassParams ::= ClassParam {, ClassParam}
ClassParam ::= [{Modifier} (val | var)]
id : ParamType [= Expr]
*Modifier ::= (**public**|**protected**|**private**)*
ClassBody ::= { [ClassBodyDecls] }
ClassMemberDecl ::= FieldDecl
| MethodDecl
| ConstructorDecl
| FunctionDecl
| ClassDef

A Class declaration has the following components (in the given order):

(i) *id* : The name of the class.
(ii) *modifier (Optional)*: This restricts the visibility of the class constructor.
(iii) *ClassParamClause*: Contains a list of formal value parameters for the default constructor. The scope of the formal parameter is the whole class declaration id.
(iv) *ExtendClause (optional)*: A List of well-formed trait names that are accessible from this class declaration.
(v) *ClassBody* : Defines the class members i.e. fields, methods, constructors and nested class definitions

9.3.1 Trait Declaration

TraitDef ::= id [TypeParamClause]
[ExtendClause] TraitBody
*ExtendClause ::= **extends** TraitTypes*
TraitTypes ::= TraitType {, TraitType}
TraitBody ::= { [MethodDecls] }

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.

9.4 Expressions

We give an overview of the expressions in the language, namely:

a. *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*.
SimpleExpr ::= new Id([Exprs])

b. *This and Super*
SimpleExpr ::= [id .] this
| [id .] super . id

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*.

c. *Method Invocation*:
Expr ::=id([Exprs])
Exprs ::= Expr {, Expr}

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*
- 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*.

d. *Blocks*
Block ::= {BlockStat ;}
BlockStat ::= Def
| {LocalModifier} TypeDef
| Expr1

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*.

e. *Assignments*
[SimpleExpr .] id = Expr
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*.

f. *If Expressions*:
if (Expr) { Expr } [else { Expr }]
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

g. *While Loop Expression*
while (Expr) { Expr }
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.

h. *Do Loop Expressions*
do { Expr } while (Expr)
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.

i. *Lambda Expression*
Expr ::='#'(Bindings) => Expr
Bindings ::= (Binding {, Binding})
Binding ::= id : Type
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 pairwise distinct. The scope of the parameters is the expression *e*. The expression must conform to the expected type of the Lambda expression.

10.0 Compiler

The compiler is written in Java. It compiles programs and generates JVM bytecodes (Lindholm and Yellin, 1999) which can execute on the JVM. The compilation is done over a number of phases. These phases include:

10.1 Lexical Analysis

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.

10.2 Parsing

The parser constructs the Abstract Syntax Tree(AST) from the token stream.

10.3 Semantic analysis

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.

10.4 Name analysis algorithm

(i) Build a global class and trait environment.
(ii) Resolve uses of types.
(iii) Check well-formedness of the class hierarchy.
(iv) Disambiguate uses of names.
(v) Resolve uses of locals and formals variables.
(vi) Resolve uses of methods and fields.

Type Analysis: The part of the compiler that does the type analysis is called a ***typechecher***. The Typechecker performs the following tasks:

(i) Determine the types of all expressions.
(ii) Check that values and variables are used consistently with their definitions and with the language semantics.

Type checking is performed bottom-up on the AST.

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, I.e., DefAssign where each local variable and every blank *val* must have a definitely assigned value before any access of its value occurs and DefUnassign where every blank *val* variable must be assigned at most once; it must be definitely unassigned when an assignment to it occurs.

Uncurry/Closure Conversion: 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

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.

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.

Bytecode Generation

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

10.5 Related Work

The concept of traits were first introduced by Smalltalk, but the version used here is based on (Scharli et al, 2003; Scharli *et al*., 2002).

The syntax closely follows that of Scala (Odersky *et al*., 2008)

Object based encapsulation is an extension of the one implemented in Scala (Odersky *et al*., 2008) and also an adaptation of the one in Newspeak.

Elimination of static state is partly an adaptation of work by Bracha (Bracha, 2010) in which he does the same for his *dynamically typed language*. In this paper, the same idea is applied to *statically typed language*.

Uniform object model was inspired by Smalltalk (Goldberg and David, 1985), Newspeak(Bracha, 2010) and Scala (Odersky *et al*., 2008).

In implementing the compiler, a lot was learnt from browsing *javac* compiler source code from the OpenJDK project and Fortress project.

11.0 Conclusion and Further 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, i.e., a formalization of the design of the language along with proofs of type safety and implementation of a production quality compiler.

Acknowledgment
Special thanks to both of my supervisors Dr.Waweru Mwangi and Dr. John Mathenge Kanyaru for numerous helpful discussions and feedback.

References

Ackerman, W. B.and Jack B. D. (1979). VAL: a Value oriented Algorithmic Language.

Ancona, D., Lagorio, G. and Zucca, E. (2000). Jam --- A Smooth Extension of Java with Mixins. **In** ECOOP00 European Conference on ObjectOriented Programming. Springer, pp. 154-178.

Cambridge, MA: Massachusetts Institute of Technology, Laboratory for Computer Science.

Bracha, G. (2010). On the Interaction of Method Lookup and Scope with Inheritance and Nesting. Language.

Bracha, G.and Cook, W. (1990). Mixin-Based Inheritance. In ACM Sigplan Notices. ACM, pp. 303-311.

Barendregt, H. P. (1981). The Lambda Calculus its Syntax and Semantics, North-Holland.
Barnes, J. (2008). Ada 2005 Rationale, Springer.

Bacon, D. F.(2003). Kava: a Java dialect with a uniform object model for lightweight classes. Concurrency and Computation: *Practice and Experience,* **15**(35), pp.185-206.

Boehm, B. W., Brown, J. R. and Lipow, M.(2008). Quantitative evaluation of software quality.

Cielecki, M., Fulara, J. and Jakubczyk, K. (2006). Propagation of JML non-null annotations in Java programs. of programming in Java.

Cardelli, L. (1996). Type systems. *ACM Computing Surveys*, **28**(1), pp.263-264.

Clocksin, W .F. and Mellish, C. S. (1981). Programming in Prolog, Springer-Verlag.

Chivers, I. and Sleightholme, J. (2005). Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003 and 77. Springer: London, U.K. Springer, 2006. Print.

Dahl, O. J., Myrhaug, B. and Nygaard, K., (1968). SIMULA 67. Common Base Language.

Goldberg, A, and David R. (1985). Smalltalk-80: The Language and Its Implementation. Reading, MA: Addison-Wesley.

Gosling, J. (2005). The Java Language Specification, Third Edition, Addison Wesley.

Halloway, S. (2009). Programming Clojure S. Davidson Pfalzer, ed., Pragmatic Bookshelf.

Harrison, M.(1967). The Programming Language LISP: Its Operation and Applications, MIT Press.

Kernighan, B. W. and Ritchie, D. M.(1978). The C Programming Language, Prentice Hall.
Milner, R., Tofte, M. & Harper, R.(1990). The Definition of Standard ML, MIT Press.

Odersky, M. and Zenger, M.(2005). Scalable component abstractions. ACM SIGPLAN Notices, **40**(10), p.41.

Odersky, M. (2006). An Overview of the Scala Programming Language, Second Edition.

O'Sullivan, B., Goerzen, J. and Stewart, D. (2008). Real World Haskell, O'Reilly Media.

Scott, M. L. (2000). Programming language pragmatics, Morgan Kaufmann Pub.

Stroustrup, B.(1997). The C++ Programming Language, Addison-Wesley.

Lindholm, T. and Yellin, F. (1999). The Java Virtual Machine Specification, Addison-Wesley.

Rossum, G. V. and Drake, F. L. (2010). Unicode HOWTO. *History*, **172**(7), p.10.

Scharli, N. et al. (2003). Traits: Composable units of behaviour. ECOOP 2003–Object-Oriented Programming, p. 327–339.

Schärli, N. (2002). Traits: Composable Units of Behavior. Technical Report, 2743 (November 2002), pp. 248-274.

Nutter, C. (2010). Using JRuby, Pragmatic Programmers.

Pierce, B. C. (2002). Types and Programming Languages, The MIT Press.

Taivalsaari, A. (1996). On the notion of inheritance. ACM Computing Surveys, 28(3), pp.438-479.

Borning, A. H. and Ingalls, D. H. H.(1982). Multiple Inheritance in Smalltalk-80. In Proceedings at the National Conference on Al. pp. 234-237.

Keene, S. E. (1989). Object-Oriented Programming in Common-Lisp, Addison Wesley.

Meyer, B. (1997). Object-Oriented Software Construction, Second Edition, Prentice Hall PTR.

Schaffert, C. (1986). An Introduction to Trellis/Owl. In ACM Sigplan Notices. pp. 9-16.

Flatt, M., Krishnamurthi, S. and Felleisen, M.(1998). Classes and Mixins. In Conference Record of POPL98 The 25th ACM SIGPLANSIGACT Symposium on Principles of Programming Languages. ACM Press, pp. 171-183.

Mens, T. and Van Limberghen, M. (1996). Encapsulation and Composition as Orthogonal Operators on Mixins: A Solution to Multiple Inheritance Problems. Object Oriented Systems, 3(1), pp.1-30.

Moon, D. A., (1986). Object-Oriented Programming with Flavors. In N. Meyrowitz, ed. Proceedings of the Conference on ObjectOriented Programming Systems Languages and Applications OOPSLA. ACM Press, pp. 1-8. [52] David A. Moon. Object-oriented programming with flavors. In Proceedings OOPSLA 86, ACM SIGPLAN Notices, volume 21, pages 18, November 1986.