Proposed UML Class Diagram for Object Functional Language (SCALA)

Veena N. Jokhakar
Department of Information Communication and Technology
Veer Narmad South Gujarat University
Surat

ABSTRACT
UML, the Unified Modeling Language and entity relationship diagrams develops a design model for almost any software built using any of the object orientated programming language. Still this lacks in coverage for functional object orientated language like scala and others. This paper proposes new idea of modeling the functional languages that cover traits, mixins, linearization, singleton classes and Case Classes specifically.

Keywords
UML diagrams, functional object oriented languages, traits, linearizations, singleton classes.

1. INTRODUCTION
UML, the Unified Modeling Language, allows a design model to be constructed, viewed, developed, and implemented in a customary way at analysis and design phase. UML as blueprint is about completeness. In forward engineering, the idea is that blueprints are developed by a designer whose job is to build a detailed design for a programmer to code up[1]. The key component of system modeling, which underlies the principles of MDA—Unified Modeling Language (UML)—is used to define several kinds of diagrams, their elements and notation. In fact, UML diagrams should be considered as a way of describing the system from various perspectives: whereas a static diagram is used to represent the structure of the system, dynamic diagrams describe its behavior[2]. The class diagram, being the most common in modeling object-oriented systems, is used to model the static design view of a system. According to MDA, the automatic transition from class diagram into platform-specific software components is done by performing a model transformation, where model elements and parameters are mapped to corresponding elements and parameters in the software code.

One can model just about any type of application, running on any type and combination of hardware, operating system, programming language, and network, in UML. Its flexibility lets you model distributed applications that use just about any middleware on the market. Relationships clearly can be represented in object-oriented languages—indeed patterns have been established for the purpose[3]. Built upon fundamental OO concepts including class and operation, it's a natural fit for object-oriented languages and environments such as C++, Java, and the recent C#, but you can use it to model non-OO applications as well in, for example, Fortran, VB, or COBOL. UML Profiles (that is, subsets of UML tailored for specific purposes) help you model Transactional, Real-time, and Fault-Tolerant systems in a natural way.

UML 2.0 defines thirteen types of diagrams, divided into three categories: Six diagram types represent static application structure; three represent general types of behavior; and four represent different aspects of interactions:

Structure Diagrams include the Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram.

Behavior Diagrams include the Use Case Diagram (used by some methodologies during requirements gathering); Activity Diagram, and State Machine Diagram.

Interaction Diagrams, all derived from the more general Behavior Diagram, include the Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram.

Each of these diagrams are of used for representing the system’s different design perspective for Object Oriented System Modeling.

2. FUCNTIONAL OBJECT OREIENTED KANGUAGE (SCALA)
Nowadays, the object-oriented approach to software construction is considered the most successful methodology for software design, mainly because it makes software reuse extremely easy. On the other hand, functional programs are reputedly easier to reason about, simpler to understand, and friendlier to concurrency. Functional programming offers some very elegant tools which when combined with an object-oriented program development philosophy define a really powerful programming methodology. Scala is a multi-paradigm programming language combining features of object-oriented and functional languages. It is a pure object-oriented language in the sense that every value is an object. In contrast to Java, all values in Scala are objects (including numerical values and functions). Types and behavior of objects are described by classes and traits. Classes are extended by sub classing and a flexible mixin-based composition mechanism as a clean replacement for multiple inheritances.

Scala is also a functional language in the sense that every function is a value. Scala is extensible, as the development of domain-specific applications often requires domain-specific language extensions. Scala provides a unique combination of language mechanisms that make it easy to smoothly add new language constructs in form of libraries. It interoperates with Java and .NET.

However, no proper semantic concept or representation for language specific features of languages like scala is present in UML. Hence this paper proposes UML class diagram representations for traits, mixins, linearization, single Tone Classes, Case Classes, Parameterized class with variances and relationship between class or trait and companion objects.

In the paper, we present a proposed UML Class diagram model for Functional Object Oriented Programming languages such as Scala. The paper has been organized as follows: Section 2 Related Work, Section 3 Shows the proposed model and Section 4 concludes the paper.

3. RELATED WORK

Modeling languages like UML [6] and ER Diagrams [7] provide associations and relationships as core abstractions.

Meike Massimow in his thesis [4] added stereotypes for trait mixins and other scala specific elements.

The addition made by him were for Attributes with access specifier, var, lazy variables, Traits , mixins, Class and Genericity. Generic types of polymorphic methods were shown as additional parameter list in angle brackets (operation).Traits are were treated same as abstract classes, however, with a stereotype "Trait".Abstract attributes and methods were presented in italics (attribute2, operation2). A dependency arrow with the stereotype "requires" was available (Trait3, attribute3). However for self-referenced types, the stereotype "Self" is used (Trait4). To represent one trait inherited another trait, use a dashed inheritance arrow (Trait2). This type of arrow is also used when a one-class traits mixed . Singleton objects are represented as classes and the stereotype "singleton" is added.

Though the above stated model try to model languages, but none of the above show precise representation of linearization order, parameterized classes, singleton classes and companion objects.

4. PROPOSED MODEL

We propose a model for traits, mixins, linearization, singleton classes, Case Classes and Parameterized class with variances.

Traits differ from abstract classes as an object of trait can be created, nor can they be said as similar to interfaces of java, as Scala allows traits to be partially implemented; i.e. it is possible to define default implementations for some methods. In contrast to classes, traits may not have constructor parameters, although they are used to define object types by specifying the signature of the supported methods.

Here is an example:

```
trait Equality {
  def isEqual(x: Any): Boolean
  def isNotEqual(x: Any): Boolean = !isEqual(x)
}
```

Classes and traits both can use with clause to inherit from other traits.

This trait consists of two methods isEqual and isNotEqual. While isEqual does not provide a concrete method implementation, method isNotEqual defines a concrete implementation. Consequently, classes that integrate this trait only have to provide a concrete implementation for isEqual. Figure 1 shows the trait representation.

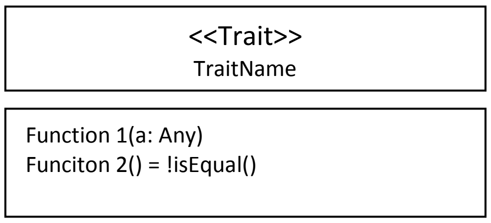

Figure 1: trait representation

In order to allow reuse of compiled classes and to ensure well-defined behavior, the linearization must satisfy the following rules:

The linearization of any class must include unmodified the linearization of any class (but not trait) it extends.

The linearization of any class must include all classes and mixin traits in the linearization of any trait it extends, but the mixin traits need not be in the same order as they appear in the linearization of the traits being mixed in.

No class or trait may appear more than once in the linearization.

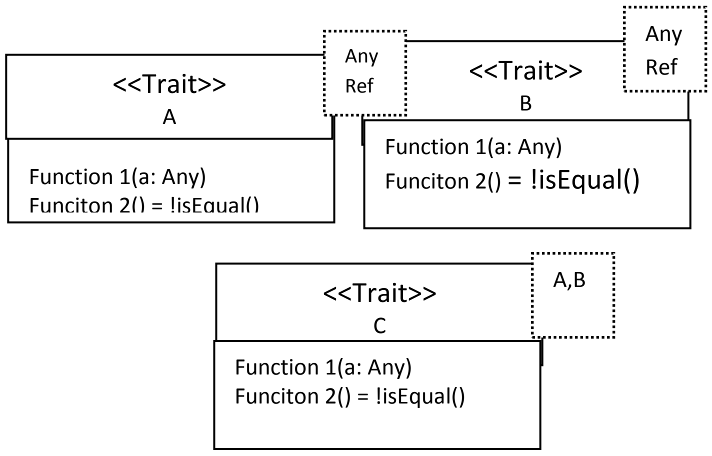

Figure 2: Linearization Order

Figure 2 shows the notation for linearization Order of inheritance. In multiple inheritance, a class can have multiple superclasses, all of which appear exactly once in the inheritance graph. With traits/mixins, each class has exactly one superclass (or supertrait), but that trait can appear in multiple different places in the inheritance graph. Hence figure two shows a way for representation of such type of inheritance and linearization.

Scala supports the notion of case classes. Case classes are regular classes which export their constructor parameters and which provide a recursive decomposition mechanism via pattern matching. Figure 3 shows the case classes.

Example below shows a class hierarchy which consists of an abstract super class Term and three concrete case classes Var, Fun, and App.

```
abstract class Term
case class Var(name: String) extends Term
case class Fun(arg: String, body: Term) extends Term
case class App(f: Term, v: Term) extends Term
```

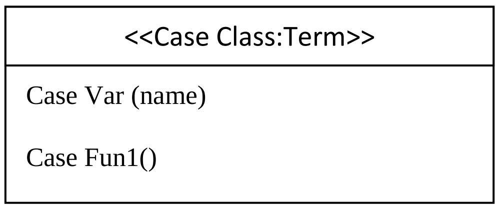

Figure 3: Case Class

Scala has built-in support for classes parameterized with types. Such generic classes are particularly useful for the development of collection classes.

```
class Stack[T] {
  var elems: List[T] = Nil
  def push(x: T) { elems = x :: elems }
  def top: T = elems.head
  def pop() { elems = elems.tail }
}
```

Scala supports variance annotations of type parameters of generic classes. In contrast to Java 5 variance annotations may be added when a class abstraction is defined, whereas in Java 5, variance annotations are given by clients when a class abstraction is used.

Type defined by the class Stack[T] is subject to invariant subtyping regarding the type parameter. This can restrict the reuse of the class abstraction.

```
class Stack[+A] {
  def push[B >: A](elem: B): Stack[B] = new Stack[B] {
    override def top: B = elem
    override def pop: Stack[B] = Stack.this
    override def toString() = elem.toString() + " " +
                          Stack.this.toString()
  }
}
```

The annotation +T declares type T to be used only in covariant positions. Similarly, -T would declare T to be used only in contravariant positions. For covariant type parameters we get a covariant subtype relationship regarding this type parameter. Figure 4 shows the parameterized Classes.

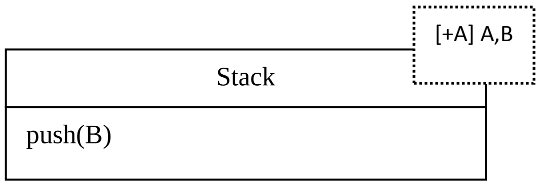

Figure 4: Parameterized Classes

Singleton Classes are such whose only one instance exists. They are named same as their class name. Figure 5 shows the singleton classes.

```
object HelloWorld {
def main(args: Array[String]) {
println("Hello, world!")
}
}
```

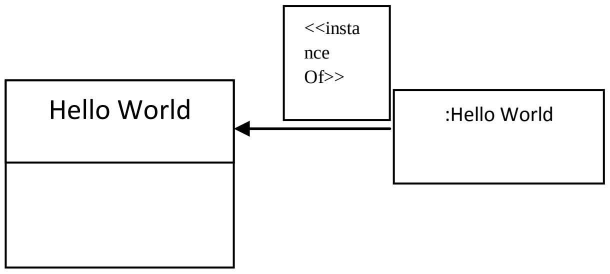

Figure 5: Singleton classes

5. CONCLUSION

Current modeling methods like UML and ER are explicitly used for OO programming. This paper proposes a very precise modeling of Scala traits/classes, linearization order , Singleton classes and companion objects, and case classes. This work can be extended further for representation of closures, Scala type members and class constructor parameter bounds, presentation of linearization with generalizations and aggregation.

6. REFERENCES

[1] Martin Flower,UML Distilled Thrid Edition, A brief Guide to Stand Object Modeling Language.

[2] Oksana Nikiforova1, Janis Sejans2, Antons Cernickins3, Role of UML Class Diagram in Object-Oriented Software Development , Scientific Journal of Riga Technical University Computer Science. Applied Computer Systems , DOI: 10.2478/v10143-011-0023-4 , Vol 44

[3] Gavin Bierman, Alisdair Wren, First-class relationships in an object-oriented language, Microsoft Research, Cambridge, University of Cambridge Computer Laboratory, FOOL 2005 15 January 2005, Long Beach, California Copyrightc 2005 ACM

[4] Meike Massimow, Evaluierung des Einsatzes von Scala bei der Entwicklung für die Android-Plattform, thesis, University of Applied Sciences,Feb 2009

[5] Eric Allen, Comparison of Object-oriented and Functional Programming for Code Generation, April 21st, 2010

[6] I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process*. Addison-Wesley, 1999

[7] P. P.-S. Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.