**An UML Virtual Machine based on AST execution**

Software Engineer Bruno Buzzi Brassesco – Reviewer VM expert Eliot Miranda

[[1]](#footnote-1)

# Abstract

An UML Virtual Machine (UVM) is a program that executes code based on an UML Class diagram (and an UML State diagram[[2]](#footnote-2)). The code to execute can have any representation like bytecodes, AST or other. The UVM analyze the code representation and creates an execution context in order to perform the code execution.

The UVM has to have the ability to create class instances based on the UML Class diagram and set and get instances’ attributes. Also it has to handle in some way the instantiation of UML Class relations (this UVM implement instances creation and update through UVM primitives and traits).

The code execution performed by the UVM has different types of restrictions defined on the UML Class diagram. For example it is forbidden to assign a string value to an integer attribute of a class.

Classes’ relations are defined by multiplicity on the UML Class diagram. Class A may have a 1:N relation with class B then an instance of class A it will have N instances of class B and each class B it will have only an instance of class A. The UVM has to do some type checks when class instances are being composed. These type checks has to include inheritance of the model too.

To summarize the characteristics of an UVM:

1. A way to import UML Class diagrams.
2. Ability to instantiate the UML Class diagram.
3. Basic building blocks to create complex operations in 4.
4. Ability to create code and code representation.
5. Ability to create context execution to perform code execution.

In order to support these five points it is very desirable to have a development environment to perform each activity.

# INTRODUCTION

In the present case the implemented UVM comply with these five points and there is also a development environment to create simulations.

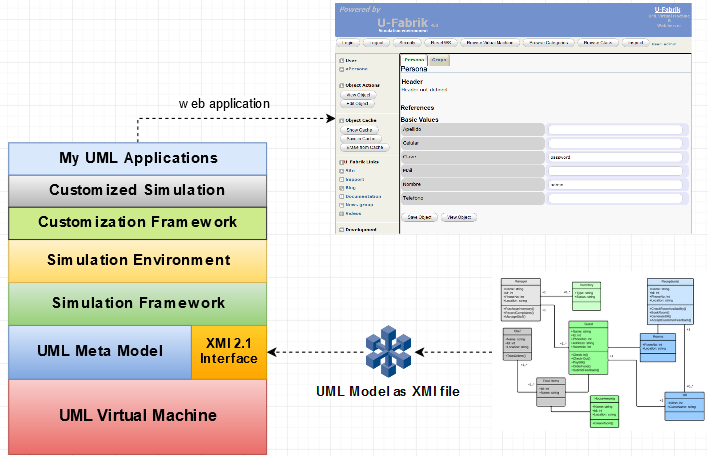
1. UML Class diagrams are imported through XMI 2.1 interface used by any UML Case tool such as Visual Paradigm, Enterprise Architect and so on ...
2. The UML Class diagram instantiation is supported by a powerful Meta Model.
3. The UVM has a basic hierarchy of classes to allow the creation of much more complex functionality.
4. The code representation is done with an AST (abstract syntax tree) using a Smalltalk Parser to transform a text into an AST.
5. Execution Context is implemented as Smalltalk classes and it’s at the core of the UVM.

The UVM was named U-Fabrik and was totally implemented in [Dolphin Smalltalk](https://github.com/dolphinsmalltalk/Dolphin) 7.0 and the code can be found at:

<https://github.com/brunobuzzi/U-Fabrik>

# U-Fabrik Architecture

The following image depict a general architecture of the whole U-Fabrik environment including the UML Virtual Machine.



The XMI 2.1 interface used to import XMI files from Enterprise Architect®, Visual Paradigm®, and so on. The Meta Model is used to represent all elements contained in the XMI file. The UML Virtual Machine to simulate behavior through message sending. On the top of the UML Virtual Machine it has a Graphical Environment for prototyping and simulation. The default web prototype is done automatically (after the XMI installation). The Customization Framework combines the Simulation Framework and Simulation Environment to achieve customization through method implementations and drag & drop operations.

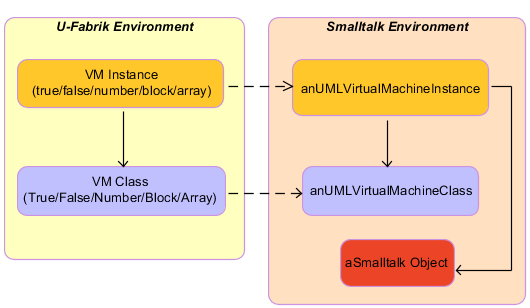
# U-Fabrik Virtual Machine (UVM)

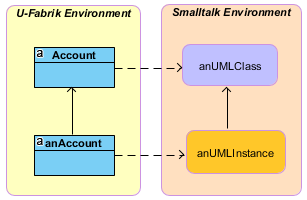
In U-Fabrik Virtual Machine objects are wrapped around Smalltalk objects. Each UVM object is an instance of Smalltalk class UMLVirtualMachineInstance. This class has an instance variable named “smalltalkObject” which holds the actual Smalltalk object.

All basic classes of the U-Fabrik Virtual Machine are instances of UMLVirtualMachineClass and are instantiated as instances of UMLVirtualMachineInstance (Smalltalk class) where “smalltalkObject” points to the actual Smalltalk object.

These classes includes: Object, Block, Boolean (True and False), Collection, Array, Dictionary, Interval, OrderedCollection, Date, Error, Exception Warning, Number, String, Time, TimeStamp, TreeModel and so on.

The following image depict how U-Fabrik objects are connected to *real Smalltalk objects*:

 In the U-Fabrik Environment instances of UML Class diagram are instances of Smalltalk class UMLInstance and its class is an instance of UMLClass:



# Virtual Machine Primitives

U-Fabrik Virtual Machine primitives interact with Smalltalk objects in order to setup U-Fabrik objects. Generally speaking a primitive take a U-Fabrik object (in some cases extract the Smalltalk object such as anArray) and perform some operation that update the state of the objects in the execution context. The operation can be performed in U-Fabrik or Smalltalk realm depending on the primitive.

Virtual Machine primitives is where Smalltalk and U-Fabrik environment meet each other.

Some Virtual Machine primitives take advantage of the parallelism between both environments and are used to perform different behavior such as <primitive: 101>. The method source in Object class for ~~, ~=, =, == it is the same a call to <primitive: 101>. This primitive is used extensively through the system. OrderedCollection methods at: and asArray are also a call to <primitive: 101>. This primitive basically extract the Smalltalk object from the U-Fabrik object and extract the method name and method arguments from the execution context and execute that in Smalltalk environment. In the final step convert the result to U-Fabrik object.

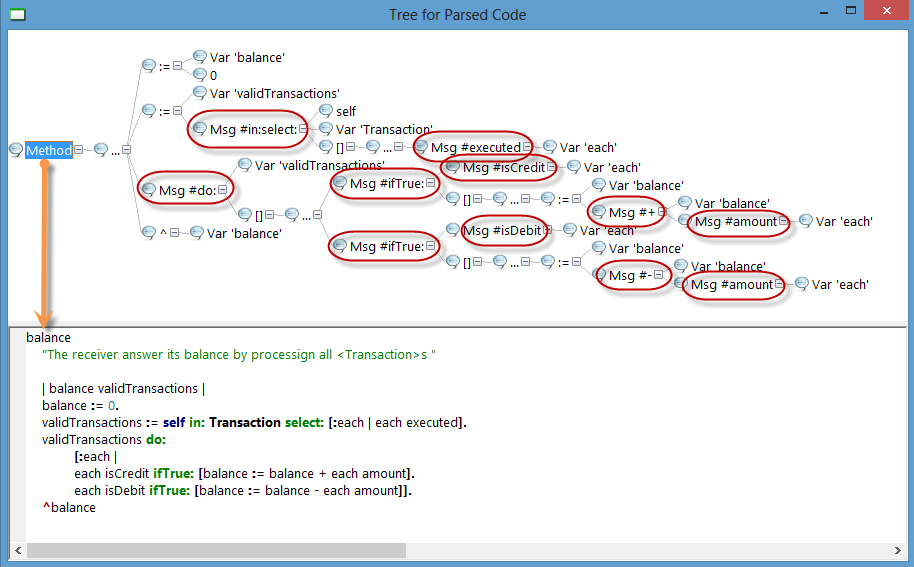
Today there are around 96 primitives in U-Fabrik and they are the glue between U-Fabrik and Smalltalk environment.

Other primitives can be easily added to U-Fabrik and incorporate Smalltalk behavior into U-Fabrik can be easily achieved too. For example the VM Class TreeModel (in U-Fabrik environment) is a partial implementation of Smalltalk class TreeModel. Most of methods source are a call to primitives 101 or 102. Primitive 102 is very similar to 101 but it does not extract the Smalltalk object from the arguments of the method.

# code execution

The U-Fabrik Virtual Machine perform code execution using AST (abstract syntax tree) nodes of each method. For each new method source created the UVM creates an instance of UMLParsedImplementation where the AST is stored (as an instance of StMethodNode). The VM know how to process each possible node of an AST. At code level all Smalltalk subclasses of StProgram node implement the method processInsideSnapshot:context:parameters: where the first parameter is the execution context (UMLImplementationExecutionSnapshot) the second is the (U-Fabrik) receiver object (UMLInstance) and the third is an array of method arguments.

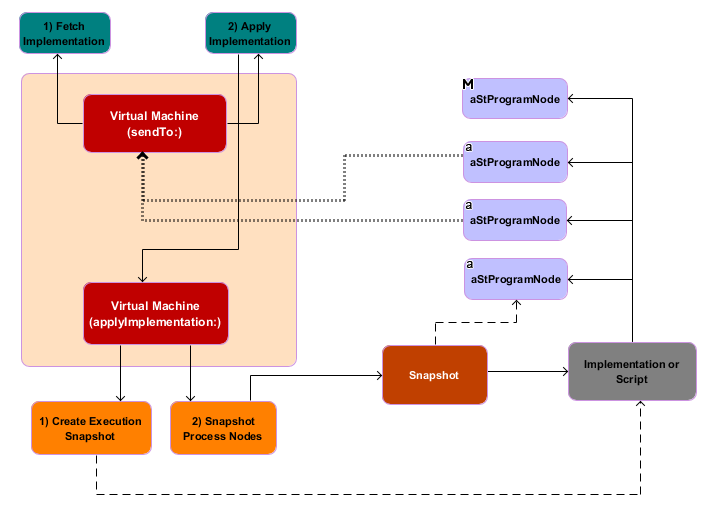
The following image show an AST for the code in bottom of the window. In red are highlighted all message send that eventually will create a new execution context.



When a message is sent to an object the following process is executed:

1. Fetch the implementation based on the receiver and the message. The implementation is an instance of (UMLParsedImplementation).
2. Apply the implementation to the receiver:
   1. Create the execution context (an instance of UMLImplementationExecutionSnapshot).
   2. Iterate over each node of the AST and process it:
      1. Update context objects (reachable objects) with new values.
      2. If there is a message send then this process start again in 1).

The below image depict this process approximately, first the VM fetch the implementation and second apply the implementation to the receiver object. Finally process each node of the AST and some nodes will trigger another message send (restart the whole process again but with a new execution context).



In each method execution the VM update the state of reachable objects within the execution context or create new objects.

# Traits

There are two different form to add behavior to U-Fabrik objects:

1. Adding methods to classes or metaclasses.
2. Creating new Traits.

Traits are composable unit of behavior that are not attached to any class. Only have required methods to be implemented for classes that use them.

The most important Trait in U-Fabrik is “UML Relations Methods”. This trait implement all methods that are used to related objects based on a UML Class diagram. It has 3 required message for the classes:

* getCollectionToOperateWith:
* getCollectionToIterateOver:
* setDirectRelationWith:

All methods are implemented at Object class as primitives so any class can use this Trait.

The primitives that implement these message use the UML Meta Model to establish what kind of relation two object can have based on restrictions in the UML Class diagram.

# Blocks

U-Fabrik implement Closures as instance of Smalltalk class UMLBlockClosureNode. Blocks are used as building blocks to create complex simulations of the UML Class diagram.

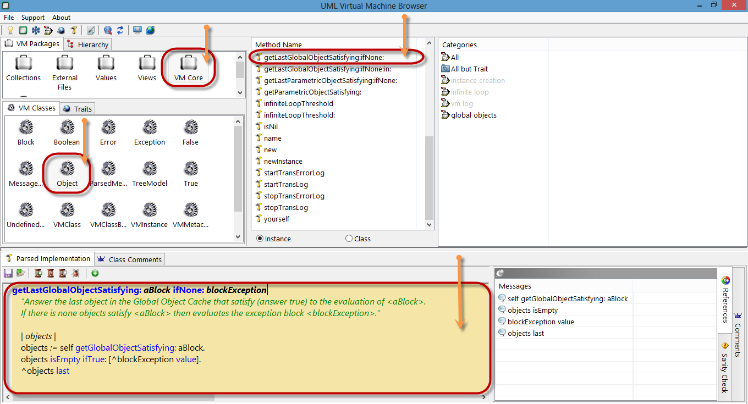
In reality all VM classes has the goal to be building blocks to create complex simulations starting with a UML Class diagram. But Blocks are also used to create the basic behavior of the VM such as Boolean implementation, the use of Exceptions and Traits implementation.

The last version U-Fabrik support re-entrant blocks and this was achieved by switching the execution context (UMLImplementationExecutionSnapshot) between each iteration of the Block.

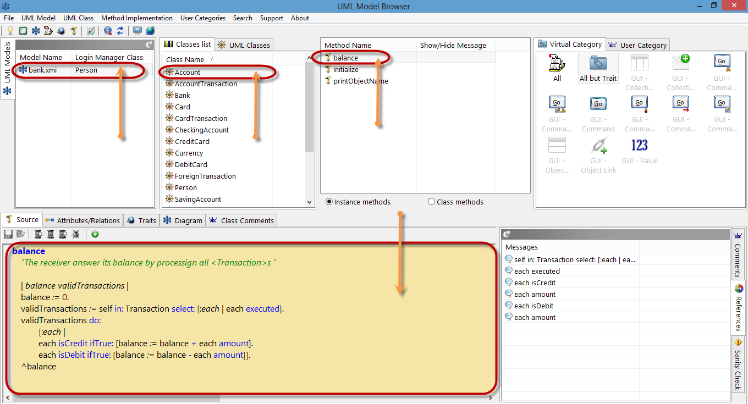
# environment

The U-Fabrik environment has several tools similar (or equal) to any Smalltalk implementation such as:

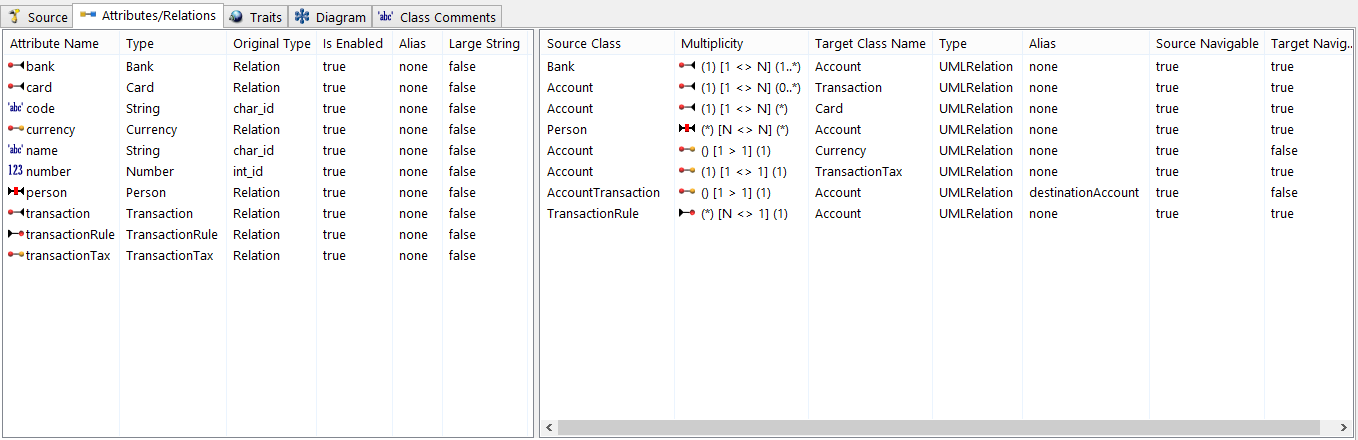
**Virtual Machine browser**



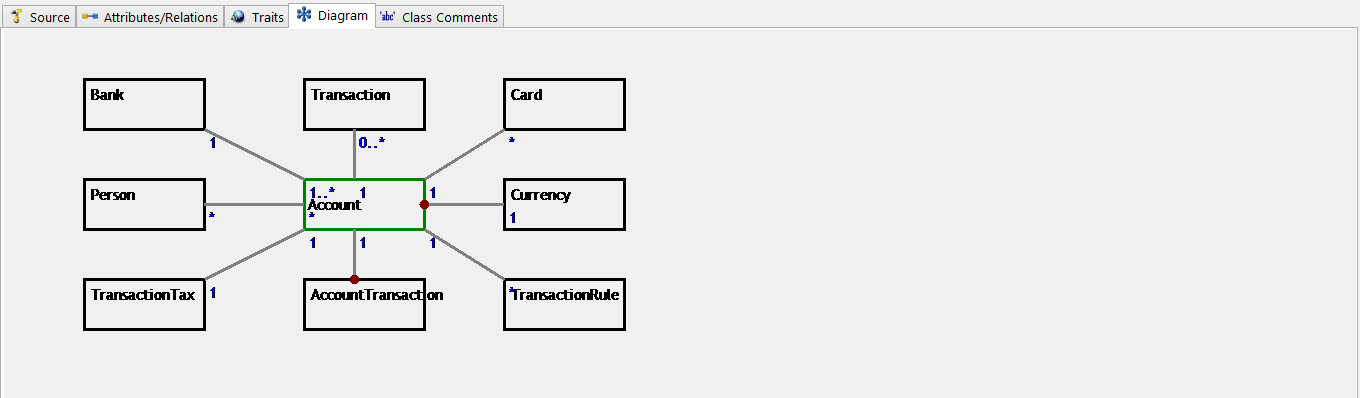
**UML Model browser**



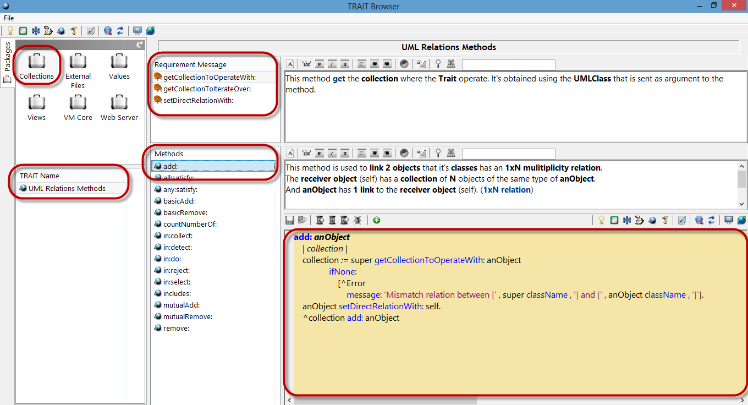
*UML Relations*



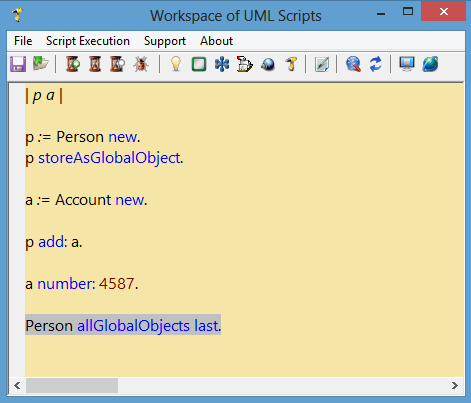
*Quick class relations view*



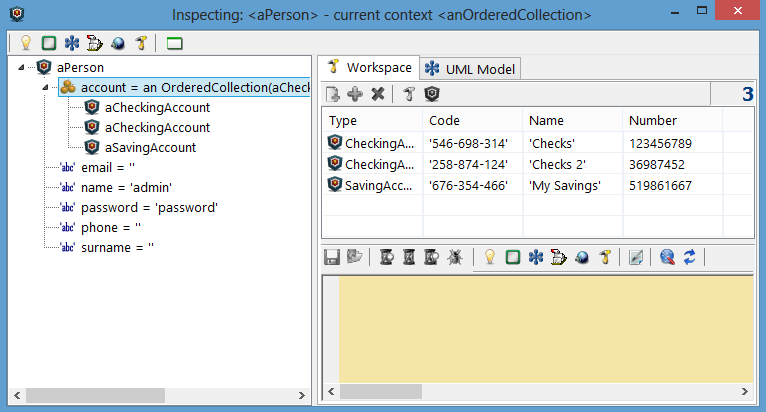
**Trait browser**



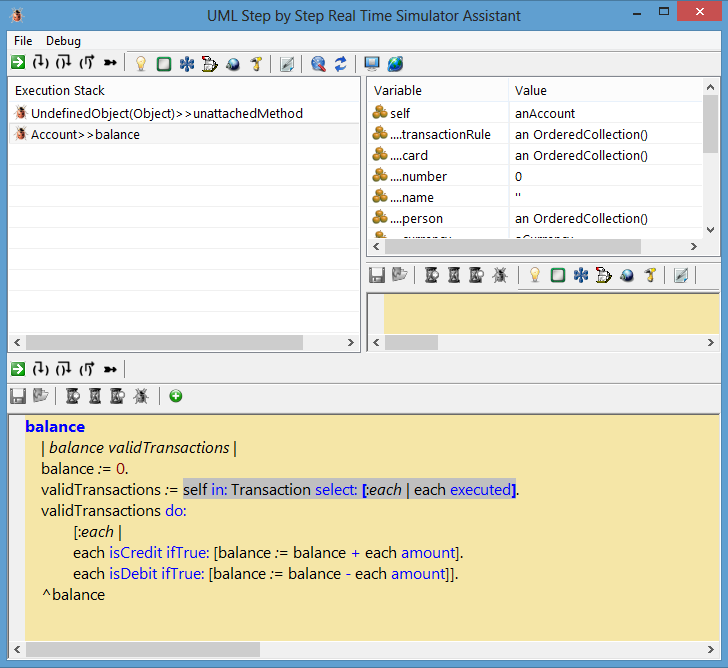
**UML Script (workspaces)**



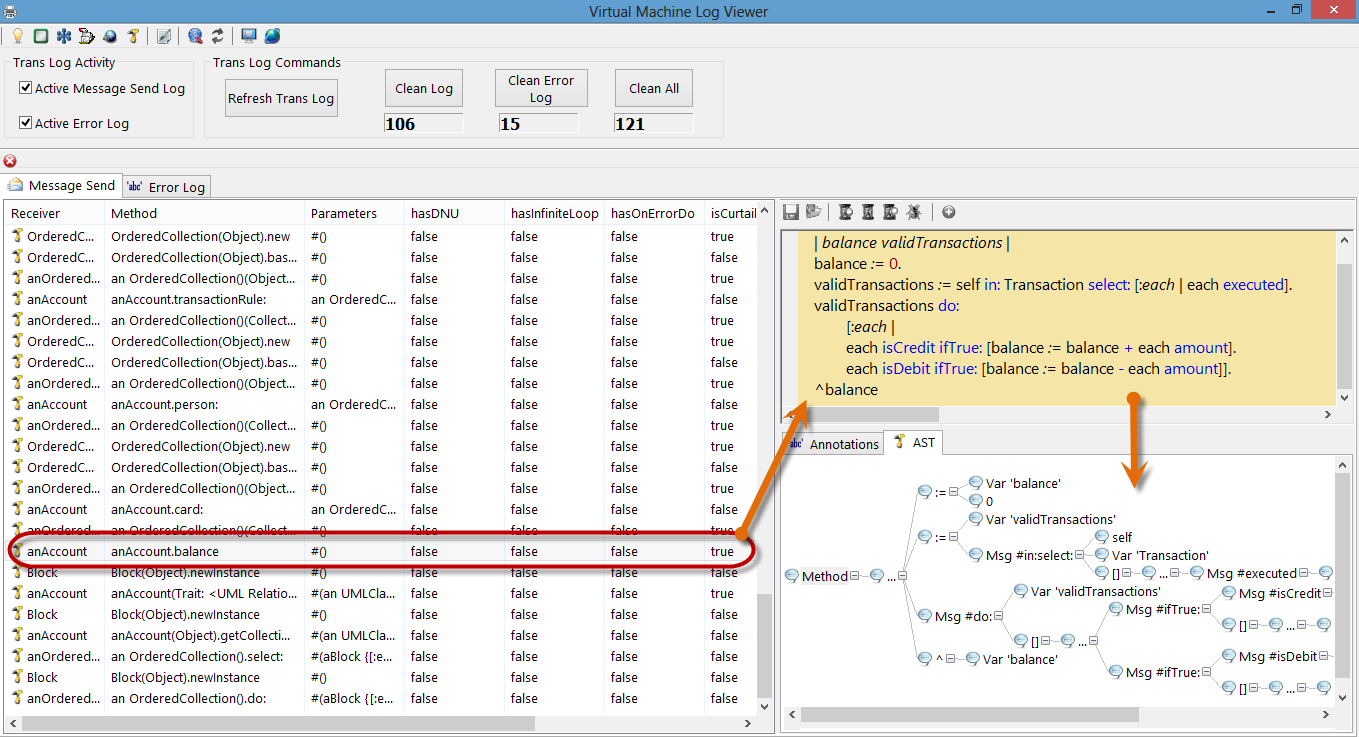
**Inspectors**



**Debugger**



**Virtual Machine log viewer**



# web simulation

All previous components of U-Fabrik environment are used to create a simulation of a UML application defined by a UML class diagram.

By default when a UML class diagram is installed in U-Fabrik a standard web simulation is already available for the user. Where each object has its own web page where:

* Simple attributes are rendered directly as simple components (strings, dates, integers, time, booleans, time stamps).
* One to one relations are rendered as web links to the related object.
* One to many and many to many relations are rendered as web list (in a separate tab).

The standard web simulation can be further customized by implementing new methods and adding (by drag & drop) these methods to special categories named “Virtual Categories”. There are eleven “Virtual Categories” to customize the standard web simulation:

**Collection Aspects** (it must answer simple values)

It will render the result of methods invocation as object aspects when the object is listed as a web list (each method will be a column).

**Collection Tab** (it must answer a collection)

It will render the result of methods invocation as new web tab with a collection of objects.

**Command Collection** (can answer any object)

It will render the result of methods invocation as web button and it will have a collection of objects as arguments of the invocation.

**Command User & Collection** (can answer any object)

The same as the previous but also pass the current user of the web application as an argument.

**Command** (can answer any object)

The same as two previous but is a web button that trigger a method without arguments.

**Command Arg** (can answer any object)

**Command External** (can answer any object)

**Command User** (can answer any object)

**Object Header** (it must answer simple values)

It will rendered the result of the methods invocation as the object header in the web page.

**Object Link** (it must answer an UML object)

It will rendered the result of the methods invocation as object link to other UML object in the web page.

**Value** (it must answer simple values)

It will rendered the result of the method invocation as read only simple web components.

If the method invocation does no answer the expected type then a web text is displayed describing the error.

# Conclusion

Appendix A: Why and how UVM was created

A big project for a government (in 2009) with 10 functional analyst (from company A) and 20 programmers (from company B) to implement a system with 400 classes or so. By contract the functional analysts (company A) have to produce an UML Model (in Enterprise Architect) and based on the UML Model create Use Cases (as Word Documents) to send it to the programmers (the software factory - company B).

This methodology had a lot of problems, most Use Cases were rejected by the Software Factory because different errors in the UML model. Errors types: referencing a missing class, a missing attribute, missing multiplicity, incompability with other part of the model (when more than one functional analyst were responsible for a Use Case), to name just a few problems.

In order to improve this situation it was visualized that another tool was needed.

The first version was a simple application to import XMI files from Enterprise Architect and display classes, relations and attributes to detect missing UML elements. The rejection of Uses Cases felt dramatically (around 80%).

The second version allow the creation of UML instances in order to dynamically visualize how elements interact with each other. This helped to refine the UML Class diagram and improve the Uses Cases creation.

The third version added a web interface to show UML instances as a Web Application enabling a much better communication with end users to break down requirements.

The fourth version introduced a mayor change, the support for method execution including Blocks. With this version a Use Case could be executed before been written as Word Document. This assured 100% of accuracy in Uses Cases and saved a lot of time and money.

This last version was truly an UML Virtual Machine that was named U-Fabrik.

U-Fabrik was implemented in Dolphin Smalltalk 6 in 2009 and it was migrated to Dolphin Smalltalk 7.0 in 2018.

Acknowledgment

References

1. [↑](#footnote-ref-1)
2. If the UML State diagram is defined as classes in the UML Class diagram then the UML Virtual Machine can simulate the State Machine. For more details: <https://github.com/brunobuzzi/U-Fabrik/wiki/State-Machine-Import> [↑](#footnote-ref-2)